DOCUMENT RESUME

ED 210 503 CE 030 775

TITLE	Microcomputer Operations. Energy Technology Series.
INSTITUTION	Center for Occupational Research and Development, Inc., Waco, Tex.; Technical Education Research Centre-Southwest, Waco, Tex.
SPONS AGENCY	Office of Vocational and Adult Education (ED), Washington, D.C.
BUREAU NO	498AH80027
PUB DATE	Dec 81
CONTRACT	300-78-0551
NOTE	259p.; For related documents see CE 030 771-789 and ED 190 746-761.
AVAILABLE FROM	Center for Occupational Research and Development, 601 Lake Air Dr., Waco, TX 76710 ($2.50 per module; $17.50 for entire course).
EDRS PRICE	MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS	Adult Education; Behavioral Objectives; Computer Storage Devices; Course Descriptions; Courses; Data Collection; *Energy; *Energy Conservation; Glossaries; Laboratory Experiments; Learning Activities; Learning Modules; *Microcomputers; Postsecondary Education; *Power Technology; *Programing; Programing Languages; *Technical Education; Two Year Colleges
IDENTIFIERS	BASIC Programing Language

ABSTRACT

This course in microcomputer operations is one of 16 courses in the Energy Technology Series developed for an Energy Conservation-and-Use Technology curriculum. Intended for use in two-year postsecondary technical institutions to prepare technicians for employment, the courses are also useful in industry for updating employees in company-sponsored training programs. Comprised of seven modules, the course covers the operation and programming of microcomputers. Focuses include general concepts (computer codes, binary arithmetic, computer parts), their application to typical energy-related data-gathering and control problems, disk-based systems, energy conservation, and BASIC programming. Written by a technical expert and approved by industry representatives, each module contains the following elements: introduction, prerequisites, objectives, subject matter, exercises, laboratory materials, laboratory procedures (experiment section for hands-on portion), data tables (included in most basic courses to help students learn to collect or organize data), references, and glossary. Module titles are Computer Codes, Microcomputer Architecture, Microcomputer Applications, Disk-Based Operating Systems, Energy Applications of Microcomputers, Introduction to BASIC, and BASIC Programming. (YLB)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED210503

MICROCOMPUTER OPERATIONS

CENTER FOR OCCUPATIONAL RESEARCH AND DEVELOPMENT
601 LAKE AIR DRIVE
WACO, TEXAS 76710

CE030775

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

This document has been reproduced as received from the person or organization originating it.
Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

"PERMISSION TO REPRODUCE THIS MATERIAL IN MICROFICHE ONLY HAS BEEN GRANTED BY

D. M. Hull

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

DEC. 1981

PREFACE

ABOUT ENERGY TECHNOLOGY MODULES

The modules were developed by TERC-SW for use in two-year postsecondary technical institutions to prepare technicians for employment and are useful in industry for updating employees in company-sponsored training programs. The principles, techniques, and skills taught in the modules, based on tasks that energy technicians perform, were obtained from a nationwide advisory committee of employers of energy technicians. Each module was written by a technical expert and approved by representatives from industry.

A module contains the following elements:

Introduction, which identifies the topic and often includes a rationale for studying the material.

Prerequisites, which identify the material a student should be familiar with before studying the module.

Objectives, which clearly identify what the student is expected to know for satisfactory module completion. The objectives stated in terms of action-oriented behaviors, include such action words as operate, measure, calculate, identify and define, rather than words with many interpretations, such as know, understand, learn and appreciate.

Subject Matter, which presents the background theory and techniques supportive to the objectives of the module. Subject matter is written with the technical student in mind.

Exercises, which provide practical problems to which the student can apply this new knowledge.

Laboratory Materials, which identify the equipment required to complete the laboratory procedure.

Laboratory Procedures, which is the experiment section, or "hands-on" portion of the module (including step-by-step instruction) designed to reinforce student learning.

Data Tables, which are included in most modules for the first year (or basic) courses to help the student learn how to collect and organize data.

References, which are included as suggestions for supplementary reading/viewing for the student.

CONTENTS

Preface

Module MO-01 Computer Codes

Module MO-02 Microcomputer Architecture

Module MO-03 Microcomputer Applications

Module MO-04 Disk-Based Operating Systems

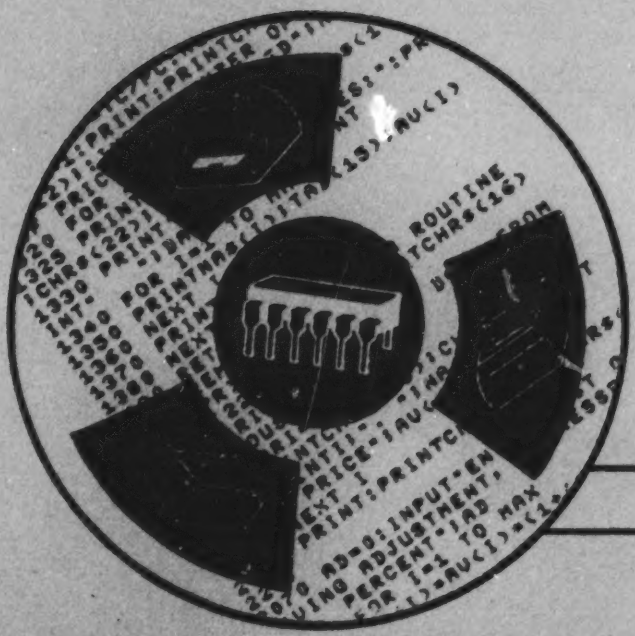
Module MO-05 Energy Applications of Microcomputers

Module MO-06 Introduction to BASIC

Module MO-07 BASIC Programming

ENERGY TECHNOLOGY

CONSERVATION AND USE



MICROCOMPUTER OPERATIONS

MODULE MO-01

COMPUTER CODES

CENTER FOR OCCUPATIONAL RESEARCH AND DEVELOPMENT

© Center for Occupational Research and Development, 1981

This work was developed under a contract with the Department of Education. However, the content does not necessarily reflect the position or policy of that Agency, and no official endorsement of these materials should be inferred.

All rights reserved. No part of this work covered by the copyrights hereon may be reproduced or copied in any form or by any means — graphic, electronic, or mechanical, including photocopying, recording, taping, or information and retrievals systems — without the express permission of the Center for Occupational Research and Development. No liability is assumed with respect to the use of the information herein.

CENTER FOR OCCUPATIONAL RESEARCH AND DEVELOPMENT

INTRODUCTION

In this course the student is introduced to the fundamentals of microcomputer operations.

This first module covers the subject of computer codes. Humans speak one language while computers "speak" another - called "machine language" - and communications between humans and machine cannot be accomplished until the human language is "coded" into machine language, and vice-versa. Computer language codes are based on numbering systems such as "binary," (based on two numbers, 0 and 1) or "octal" (based on numbers 0 through 7). These codes are different from, but no more complicated than, the familiar "decimal" system (based on ten numbers, 0 through 9), which is commonly used.

PREREQUISITES

The student must be able to calculate the value of integer exponentials like 2^5 and use a voltmeter to measure voltages.

OBJECTIVES

Upon completion of this module, the student should be able to:

1. Distinguish between logical and analog signals.
2. State and use the TTL definition of logical signals.
3. Convert between binary, octal, hexadecimal, BCD and decimal codes.
4. Add in binary.
5. Use a table to convert between ASCII codes and alphanumeric data.

6. Safely turn on a microcomputer and use it to examine inputs and alter outputs.
7. Define the following terms:
   a. Microcomputer.
   b. Integrated circuit.
   c. Electrical code.
   d. Logical code.
   e. Bit.
   f. Threshold voltage.
   g. State of a line.
   h. Indeterminant range.
   i. Analog signal.
   j. Analog-to-digital convertor.
   k. Digital-to-analog convertor.
   l. Nyble.
   m. Byte.
   n. Alphanumeric.
   o. Program.
   p. Word.
   q. Multiple-precision data.
   r. Super words.
   s. Binary.
   t. Least significant bit.
   u. Most significant bit.
   v. Octal.
   w. Radix.
   x. Base.
   y. Hexadecimal.
   z. Kilobyte.
   aa. Megabyte.
   bb. Binary coded decimal.
   cc. Operation code.

8. Identify the following abbreviations:
   a. ADC
   b. DAC
   c. BCD
   d. LSB
   e. MSB
   f. ASCII
   g. Op-code
   h. TTL

ELECTRICAL CODES

Many questions have yes or no answers: Did you get up before eight this morning? Was it a cloudy day? Are you sitting down right now? Are you a millionaire? Is an electron heavier than a neutron? Were dinosaurs warm-blooded? The answers may not be known, but in every case there are exactly two possible answers. The answers tell something about the world; they give some information. In fact, the answer to a yes-or-no question gives the smallest amount of information, called a bit.

DIGITAL CODES

The electrical signals used inside a computer are just like the yes-or-no answers discussed above. The signals are either above or below a certain voltage called the threshold voltage. If a voltage on a particular line is above the threshold, it is referred to as being true, or high, or logical one, or simply "1". Conversely, if the voltage is below the threshold voltage, it is said to be false, or low, or logical zero, or simply "0". There is no such thing as a computer signal that has a very low voltage, or a very high voltage, or an intermediate voltage. To the computer, all voltages are either high or low, and are referred to as two states of a line. These states convey 1 bit of information. Figure 1 summarizes the various names given to the two states.

Computer circuits cannot tolerate voltages near the threshold. A given voltage near threshold might be sensed

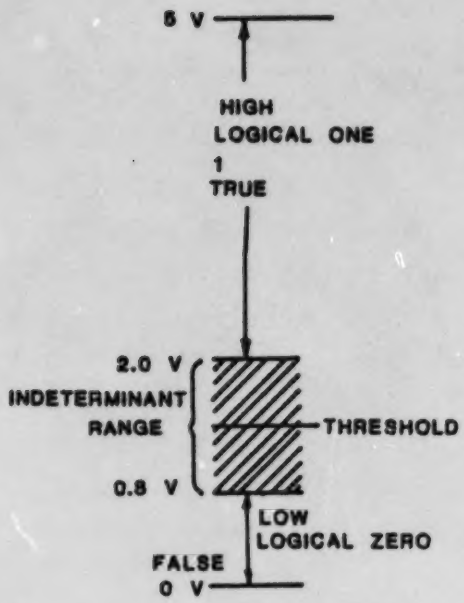


Figure 1. TTL Voltage Levels.

as logical one by one circuit and as logical zero by another. This can happen because of variations in the properties of circuits that cannot be controlled as they are manufactured. As a result, there is a kind of "no-man's land" around the threshold voltage, called the indeterminant range, which should be avoided whenever possible.

It is surprising that digital computers use such a simple electrical code. It seems strange that computers are designed for complex tasks, yet work on such a simple basis. The answer lies in speed. The two states of logical zero and logical one correspond to transistors turning on and off. In fact, circuits inside computers can turn on and off millions of times every second; this gives them the ability to do very complex things by doing a lot of simple things very fast.

Some computers use different voltage levels to correspond to the two logical states, but the most common voltage definitions are called TTL. TTL is an abbreviation for Transistor Transistor Logic and refers to a specific kind of circuit

used within the computer. For TTL, logical zero is defined as any voltage between 0 volts (0 V) and 0.8 V. The indeterminant region extends from 0.8 V to 2 V. Any voltage between 2 V and 5 V is considered to be logical one. Voltages above 5 V can damage circuits, as can voltages below 0 V. These relationships are illustrated in Figure 1.

Any TTL signal will have two states. For instance, in the computer circuits that use TTL, 3 V and 4 V are equivalent, i.e., they are both logical 1. This is an example of a digital code.

Any device using signals that have two states is called a digital device; consequently, computers using two states as their electrical codes are called digital computers. Microcomputers are one type of digital computer which shares this property with some of the largest computers. Many other devices, such as printers, teletypes, keyboards, card readers, and some laboratory instruments, communicate information over digital lines; those that do so are called digital devices, even though they are not computers.

ANALOG CODES

Another type of code, called an analog signal, has a different meaning for each voltage. For instance, the solar cell in Figure 2 generates a voltage that is related to the amount of light that falls on it; that is, more light is falling on it when it generates 1.0 V than when it generates 0.9 V. Similarly, 0.95 V indicates a light level somewhere between these values. Therefore, unlike digital signals, a characteristic of analog signals is that each voltage has a different significance. Computers that use analog signals internally are called analog computers; but

these computers are not as flexible or as powerful as the digital computers and are only used to solve certain, very special problems to which they can be applied easily.

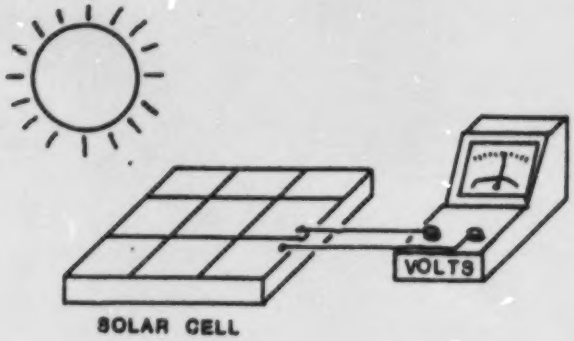


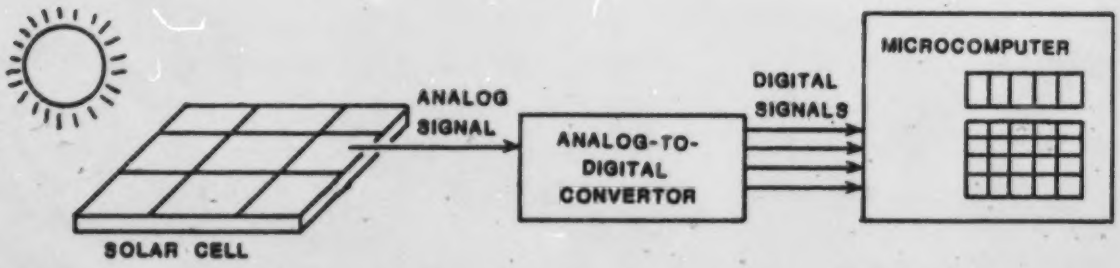
Figure 2. Analog Signal Generated by a Solar Cell.

Analog signals cannot be used directly by digital computers. For instance, to read the signal from a photocell into a digital computer as shown in Figure 3a, some means must be found to convert the analog voltage produced by the solar cell into a digital signal that can be used in the computer. This is done by an analog-to-digital convertor (ADC), which is not, strictly speaking, a part of the computer, but is an essential tool for many scientific and industrial applications of microcomputers.

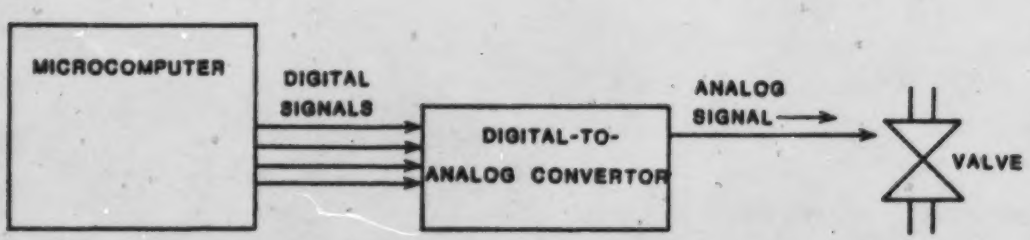
A similar situation occurs when the computer is used to set an analog signal. This might occur, for instance, in an industrial situation where the computer has control over the position of a valve as shown in Figure 3b. In response to some input, the computer is used to increase or decrease the flow through a valve. The control signal sent to the valve uses an analog code that is related in some way to the valve position. The computer cannot generate this analog code directly, but needs an interface between its digital signals

and the analog signals required by valves and other similar devices. A digital-to-analog convertor (DAC) is usually used in this situation to generate analog voltages from the digital outputs of the computer.

Both analog-to-digital and digital-to-analog convertors are found in many microcomputer applications to scientific, technical, and industrial problems. The details of how they perform their conversions and what form these conversions take are subjects for later modules. At this point it is sufficient to note the distinction between analog and digital signals, and to realize that many signals are analog, therefore some means, such as convertors, must be used to get these signals into and out of the digital world of computers.



(a)



(b)

Figure 3. Convertors Generate and Detect Analog Codes.

LOGICAL CODES

A code is something that stands for something else. For instance, a person's name is a type of code; but it is just a general code, because usually there are many other people in the world with the same name. A Social Security number is another code for a person; and since this number is assigned to only one person, it is a better code. Still, the meaning of this number alone is not clear until the significance of the code is known. The number could be a telephone number or a random sequence of digits. It is important to remember that codes have no use unless there is agreement about their meaning.

This section of the module discusses the codes used in a computer; these have meanings already given to them that must be understood if the student is to understand the computer.

At the basic level, there are various voltages which have certain simple, logical meanings. However, a single electrical line cannot convey much information. The Social Security number code, for instance, consists of nine digits and conveys much more information than a single digit could. In the same way, most codes used within a computer consist of many different lines grouped together in various logical ways. These groupings can convey much more information than single lines.

BYTES, WORDS AND SUPERWORDS

Only a very small amount of information can be conveyed in a 1-digit line which can only take one of two states. This amount of information, called a bit, is the fundamental unit of information used within a computer. Because a bit is such

a small quantity, bits are usually grouped together in larger units. In the binary code, the smallest grouping of bits together is 4 bits, which is sometimes called a nyble, or a hexadecimal digit.

There are 16 possible states of the 4 lines ($2^4 = 16$), which is much better than 2 states of a single line ($2^1 = 2$). There are many ways of assigning codes to each unique state of four lines. Of the ways that this assignment could be made, the more traditional one is called Binary Coded Decimal (BCD), which is used in Table 1. There are 16 combinations of zeros and ones taken four at a time. These are usually numbered as shown, starting with zero.

TABLE 1. THE 16 STATES OF A 4-LINE BINARY NUMBER.

Decimal Equivalent	Binary Number
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

The nyble is still too small a grouping of bits for many purposes. For instance, there are 26 letters; therefore, to represent these letters, more lines are needed than is possible in a nyble. The next larger grouping of bits is called a byte and consists of 8 bits or 2 nybles. There are 256 combinations of logical one and zero on the 8 lines ($2^8 = 256$), and as a result, a byte can represent all of the typewriter characters – upper case and lower case, numbers and letters – and still have unused combinations for special purposes.

Still, calculations that give results larger than 256 are not possible using only single bytes. For these purposes, bytes are sometimes paired together to make a word that is 16 bits long, which gives a total of 65,536 combinations ($2^{16} = 65,536$). The term "word" usually refers to 16 bits, but can refer to larger groupings. To make sure, the term "16-bit word" is often used. Even a word is not large enough for many calculations. In these cases, it is possible to string four, five or even more bytes to represent scientific or engineering data with a high degree of accuracy. This grouping is usually referred to as multiple-precision data or superwords.

Of all these groupings, a byte is probably the most important, because most computers operate one or more byte at a time. For instance, most microcomputers in use today are called 8-bit machines, which means that at the fundamental level they use 8-bit or 1-byte codes throughout. It would seem that these machines would be restricted to working with numbers that are no larger than 256; however, all computers using 8-bit or 1-byte codes have the ability to handle multiple byte words, although only 1 byte at a time. Hence, more accuracy is always possible at the price of speed.

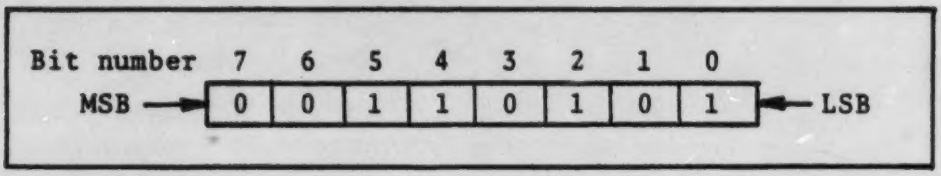
Many 8-bit microcomputers are now being replaced by 16-bit microcomputers, which retain some of the ability to work on individual bytes, but also have the capacity of using 16 bits at a time. The primary advantage of this replacement is a substantial increase in speed. User convenience and programming sophistication are secondary benefits that 16-bit machines offer due to their increased information-handling capacity.

NUMERICAL CODES

Binary

Numbers are represented inside computers in several different ways. The number 00110101 is a possible representation of the state of 8 lines, as shown in Table 2. The order is important to note. This state is different from the state 10100101, which has the same number of 0's and 1's, but in a different order. Any number represented with 0's and 1's is a binary number. The order of a group of binary bits like this is indicated by assigning each of the bits a number starting at zero, and placing the zero bit on the right with all others in ascending order toward the left. The bit on the right is called the least significant bit (LSB) and the bit on the left is called the most significant bit (MSB).

TABLE 2. A REPRESENTATION OF ONE POSSIBLE STATE OF THE 8 LINES IN A BYTE.



Many computers use simple lights to indicate the state of each of these lines. If the light is ON, the corresponding bit is 1, and if the light is OFF, the corresponding bit is 0. A long string of lights, some ON and some OFF, is sometimes quite difficult to interpret. As a result, and simply as a convenience, these lights are often bunched in groups of three or four, as shown in Table 3. (Note the similarity between Table 3 and Table 1.) Again, as a matter of convenience, the groups of three or four are often converted into digits that are easier to read and use.

TABLE 3. BINARY-OCTAL EQUIVALENTS.

Octal Digit	Binary Equivalent
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Octal

As shown in Figure 4, binary bits can be grouped by threes and converted to digits; this is called the octal numbering system. The word "octal" comes from the Greek root for eight, which reflects the fact that the three binary lines have exactly eight possible combinations. These eight combinations are assigned to numbers 0 through 7, as shown in Table 3. It would be useful for the student to memorize this table since many applications require the ability to make quick conversions in either direction.

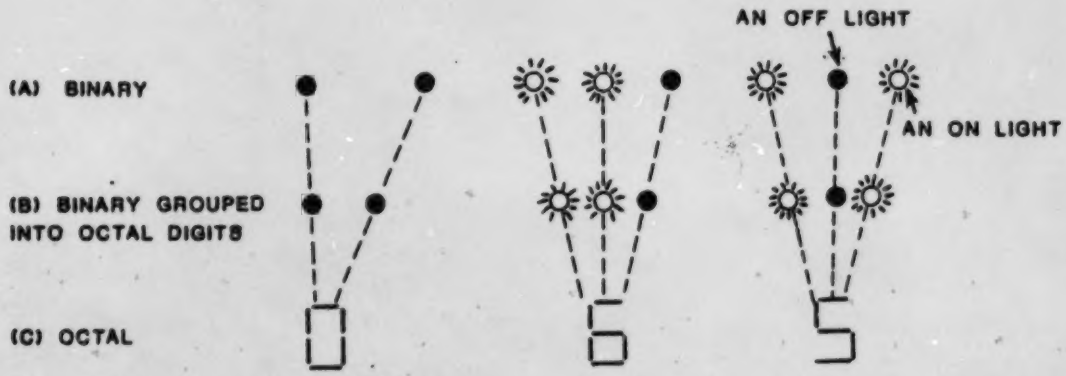


Figure 4. Binary Converted to Octal Digits.

Octal numbers should be considered as simply a convenient representation of binary numbers. For instance, the octal number 307 is simply a more convenient way of representing the binary word 11000111. It is just a different (yet equivalent) code that happens to be more convenient.

The common numbering system in everyday use is called the decimal system. The "deci" root is Greek for ten and

refers to the ten symbols, 0 through 9. Octal numbers are different from decimal numbers in some very important ways. For instance, the next octal number above 307 is not 308; this is not possible because octal digits range from 0 through 7, and 8 is not a valid octal digit. After 7, the right-hand column starts over with 0, and a "carry" command advances the next (middle) column to the next number. Thus, after 307, the next number is 310.

Radix Indication

Octal numbers are distinguished from regular decimal numbers by the addition of an 8 subscript after the octal number. Eight is the number of unique digits in the octal numbering system and is called the base. The base in the binary counting system is two; therefore, a 2 subscript is used after all binary numbers. For instance, the binary number 100 is represented as 100_2.

The base of the decimal counting system is 10, and this subscript is used to indicate a decimal number. For instance, 100_{10} is 100 in decimal. The 10 subscript is used only when there could be some misunderstanding about the base. The term radix refers to the value of the base; thus, the radix for binary is two.

Hexadecimal

Binary bits, grouped in fours, can be represented with the hexadecimal counting system. As mentioned previously, there are 16 combinations of 4 bits. The digits, 0 through 9,

can be assigned to the first 10, but new symbols are needed for the remaining six. The first six letters of the alphabet are assigned to these, as shown in Table 4. (Note that Table 4 is identical to Table 1 except for the use of letters to represent the numbers beyond nine.)

TABLE 4. HEXADECIMAL EQUIVALENTS TO THE 16 COMBINATIONS OF 4 BITS.

Hexadecimal digits	Binary Equivalent
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Hexadecimal codes are used for the same reasons as octal – as a convenience in reading and representing binary numbers. As Figure 5 illustrates, a byte, or 8 bits, can be represented by only 2 hexadecimal digits. Hexadecimal notation has an advantage over octal by requiring fewer digits, but the disadvantage of using letters can be confusing. For instance, it takes some practice to think of the letter E as fourteen, or 1110_2.

Figure 5. Two Hexadecimal Digits Can Represent 8 Bits (1 byte).

The base of the hexadecimal counting system is 16, so 16 is sometimes used as a subscript to indicate the radix. H is also used in the same way; therefore, 37_{16} and 37_H are the same hexadecimal number.

BASE CONVERSION

Four different types of numbers have been discussed: binary, octal, hexadecimal and decimal. Each is called a numbering system. To use them effectively, it is important to be able to convert from one numbering system to another. For instance, to find the 255th octal number, or the decimal equivalent of 33_8, conversions between numbering systems are required. Conversions between any two numbering systems are discussed in the following section of this module. The

simplest conversions – between binary and octal and between binary and hexadecimal – are discussed first.

BINARY/OCTAL CONVERSION

Figure 4 illustrates the principle of binary-to-octal conversion. In this example, the binary number 00110101_2 was grouped into threes to give 00 110 101. Then each group of three was converted to digits (using Table 3) to give 065_8.

The general procedure in converting any binary number to octal involves these two steps:

1) The bits are grouped in threes, starting at the right.
2) Each group is converted to a digit, using Table 3.

EXAMPLE A: BINARY-TO-OCTAL CONVERSION.	
Given:	1100111011010111_2.
Find:	The octal equivalent.
Solution:	Grouping by threes gives:
	1 100 111 011 010 111
	Converting each group gives:
	1 4 7 3 2 7
	This answer is 147327_8.

This process can be reversed for an octal-to-binary conversion. To find the binary equivalent of an octal number, each digit is simply converted in order, with leading zeros included in each digit.

EXAMPLE B: OCTAL-TO-BINARY CONVERSION.	
Given:	41776_8.
Find:	The binary equivalent.
Solution:	Converting each digit gives: 100 001 111 111 110 (Note the leading zeros.) When this is written with the numbers together, it becomes this number: 100001111111110_2.

BINARY/HEXADECIMAL CONVERSION

Conversion between binary and hexadecimal is quite similar to binary/octal conversion except that groups of four are used. Figure 5 illustrates the process. The number 00110101_2 was grouped in fours to give 0011 0101. These were converted to digits (using Table 4) to give 35_H.

The general rule for binary-to-hexadecimal conversion is as follows:

1) The bits are grouped in fours, starting at the right.
2) The groups are converted to hexadecimal digits, using Table 4.

EXAMPLE C: BINARY-TO-HEXADECIMAL CONVERSION.	
Given:	1100111011010111_2.
Find:	The hexadecimal equivalent.
Solution:	Grouping by fours gives:
	1100 1110 1101 0111
	Converting (using Table 4) gives:
	$CED7_H$.

To convert from hexadecimal to binary, the process is just reversed. Each digit is converted to binary, and then all bits are written in order. Again, the leading zeros are not dropped.

EXAMPLE D: HEXADECIMAL-TO-BINARY CONVERSION.	
Given:	$37FC0_H$.
Find:	The binary equivalent.
Solution:	Converting digit-by-digit (using Table 4) gives:
	0011 0111 1111 1100 0000.
	This gives the number:
	00110111111111000000_2.

These examples show how much more compact and less error-prone hexadecimal and octal numbers are compared to binary.

HEXADECIMAL/OCTAL CONVERSION

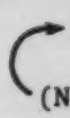
The easiest way to convert between hexadecimal and octal is to go through binary; that is, to convert an octal number to hexadecimal, the octal number is first converted to binary, which, in turn, is then converted to hexadecimal.

EXAMPLE E: OCTAL-TO-HEXADECIMAL CONVERSION.	
Given:	3701_8.
Find:	The hexadecimal equivalent.
Solution:	Converting first to binary gives:
	011 111 000 001
	Then, regrouping in fours, starting at the right, gives:
	0111 1100 0001
	Finally, converting to hexadecimal gives:
	$7C1_H$.

EXAMPLE F: HEXADECIMAL-TO-OCTAL CONVERSION.	
Given:	$FACE_H$.
Find:	The octal equivalent.
Solution:	Converting first to binary gives:
	1111 1010 1100 1110

Example F. Continued.

Regrouping in threes, starting at the right, gives:



1 111 101 011 001 110

(Note that 001 is implied here.)

Then, converting this to octal gives:

175316_8.

BINARY/DECIMAL CONVERSION

Conversions between decimal and other codes are not as easy; this cannot be done one digit at a time. The discussion below shows how to convert between decimal and binary.

Figure 6 shows the steps required to convert a binary number to decimal:

1) The binary number is written down.
2) The bits are numbered, starting with zero at the right. These numbers are called the bit numbers.
3) The decimal value of each bit is computed. This is equal to 2 raised to the bit number. For instance, bit 5 has a value of $2^5 = 32$.
4) The bit values for bits that are one are added together. The bit values for bits that are zero are not used.

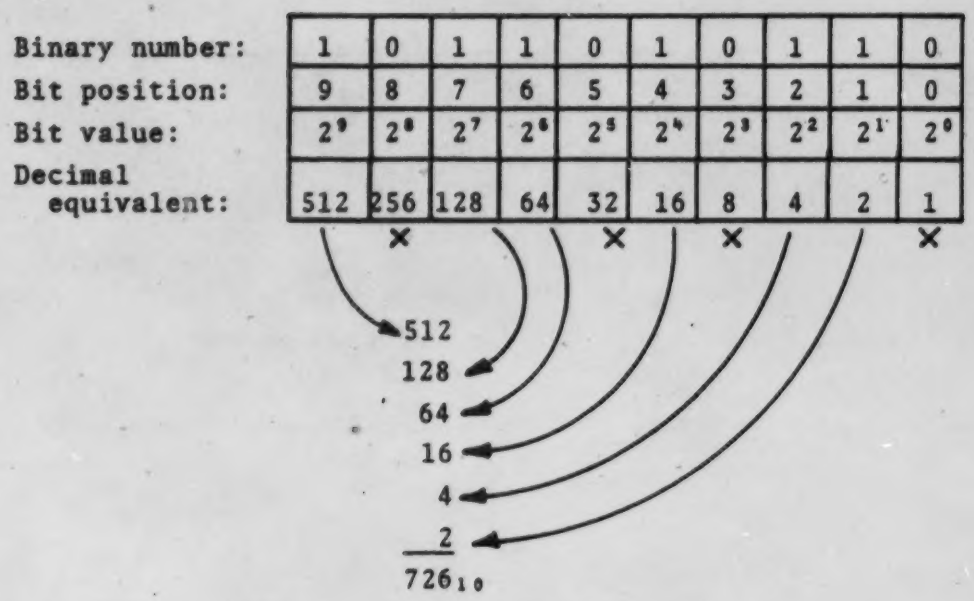


Figure 6. Steps in a Binary-to-Decimal Conversion.

The bit values are summed for all bits that are 1 in the binary number. This example shows that 1011010110_2 and 726_{10} are equivalent.

EXAMPLE G: BINARY-TO-DECIMAL CONVERSION.	
Given:	1000101_2.
Find:	The decimal equivalent.
Solution:	There are ones in bit positions 0, 2 and 6. These have bit values of 1 (as stated, $2^0 = 1$), 4 and 64. The sum of these is 69, the decimal equivalent.

The conversion from decimal to binary is quite different; it is not done digit-by-digit as it is with octal and hexadecimal. Figure 7 illustrates one correct approach.

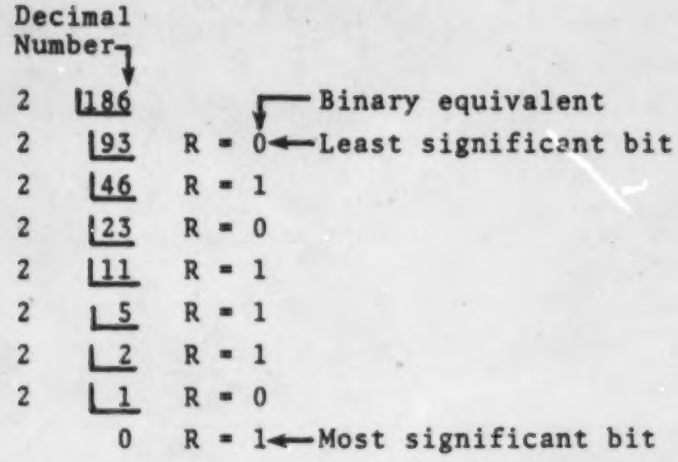

Decimal Number
2 | 186
2 | 93 R = 0 ← Least significant bit
2 | 46 R = 1
2 | 23 R = 0
2 | 11 R = 1
2 | 5 R = 1
2 | 2 R = 1
2 | 1 R = 0
0 R = 1 ← Most significant bit
Binary equivalent

Figure 7. Decimal-to-Binary Conversion.

The steps used in Figure 7 are as follows:

1) The decimal number is divided by 2 and the remainder is noted.
2) The result of the previous step is divided by 2 and again the remainder is noted.
3) Step 2 is repeated until the result is zero.
4) The binary equivalent is the sequence of remainders, with the first remainder on the right, in the LSB position.

EXAMPLE H: DECIMAL-TO-BINARY CONVERSION.		
Given:	20_{10}.	
Find:	The binary equivalent.	
Solution:	The first division gives:	10 with 0 remainder.
	The second division gives:	5 with 0 remainder.
	The third division gives:	2 with 1 remainder.
	The fourth division gives:	1 with 0 remainder.
	The last division gives:	0 with 1 remainder.
	The remainders, with the first as the least significant, give:	
	10100_2.	

The rules discussed to this point permit conversions between binary and decimal. Other conversions can always be done through binary. The conversion between octal and decimal is done through the intermediary of binary; that is, octal is converted to binary, then binary is converted to decimal.

EXAMPLE I: HEXADECIMAL-TO-DECIMAL CONVERSION.	
Given:	$3F_H$.
Find:	The decimal equivalent.
Solution:	First, convert $3F_H$ to binary, which gives:
	00111111_2.

Example I. Continued.

Then, the rules to give the decimal equivalent are used as follows:

$$2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 = 63_{10}.$$

EXAMPLE J: DECIMAL-TO-OCTAL CONVERSION.

Given: 100_{10}.

Find: Convert to octal.

Solution: First convert 100_{10} to binary. This gives:

$$1100100_2.$$

Converting this to octal gives 144_8.

BINARY ARITHMETIC

Adding binary numbers follows most of the rules normally used in adding decimal numbers. One exception is that, in binary, 1 + 1 does not equal 2, since 2 is not a binary number. Instead, 1 + 1 equals the next valid binary number, which is 10. With this one exception, long binary numbers can be added as though they were decimal, as illustrated in Figure 8.

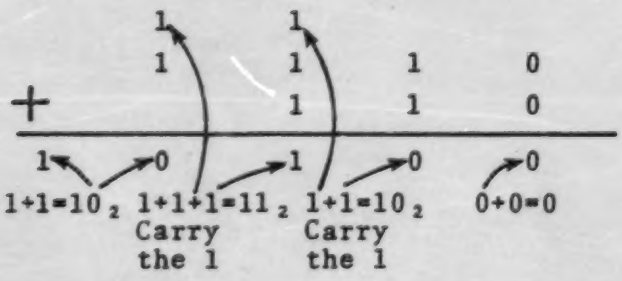


Figure 8. The Sum of 1110_2 and 110_2 Gives 10100_2.

LARGE BINARY NUMBERS

It is sometimes useful to have a shorthand for certain large binary numbers. The term "kilobyte" refers to 2^{10} = 10000000000_2 bytes. This binary number equals 1024_{10} and is sufficiently close to a thousand to deserve the metric prefix "kilo". The abbreviation K is often used to represent 1024_{10}. Thus, 64K actually means $64_{10} \times 1024_{10} = 65,536_{10}$.

Similarly, the prefix "mega", abbreviated M, is used to stand for $2^{20} = 1,098,576_{10}$. This is close enough to a million to be convenient. Thus, a disk that stores 20M bytes actually stores almost 21 million bytes ($20 \times 2^{20} = 20,971,520$).

OTHER LOGIC CODES

BCD NUMBERS

Binary-coded decimal (BCD) coding is another way of representing numbers within the computer. In this system, the binary equivalent of each decimal is coded into 4 bits. For instance, the BCD equivalent of 37 is 0011 0111. Since humans think and work in terms of decimal numbers, this coding scheme is convenient at the input and output of a computer. It is not as efficient a use of the computer's storage, however, because 1 byte can only represent the numbers from 0 to 99, which is less than half of the 255 numbers permitted by binary coding. Furthermore, binary arithmetic doesn't work with BCD code. If the computer attempted to add the BCD equivalent of 9 to 37, the result would be the following:

```
  0011 0111
+ 0000 1001
  0100 0000
```

Interpreting this as a BCD number gives 40, which is incorrect. Some computers have a decimal mode of addition, which can correct these errors by adding an additional 6 to the digits as required. In this case, 6 added to the lower digit would result in the correct BCD answer. When such a computer is operating in decimal mode, then addition of BCD data gives correct results, but addition of binary or hexadecimal coded data gives incorrect results.

ALPHANUMERIC CODES

To this point, only the encoding of numbers has been discussed. What about letters and symbols? Sometimes there is a need for a code that is large enough to encompass all 26 letters and some punctuation, plus some numbers.

Such are alphanumeric codes and a great many of them have been defined. They vary, depending upon whether upper or lower case letters are permitted and exactly what symbols are included - just as the details of typewriter keyboards vary. One of the more widely used codes is ASCII (pronounced "as-kee"). ASCII is important because it is the code most often used by terminals and printers. The letters are an acronym for American Standard Code for Information Interchange.

This code includes 128 possible symbols and characters which are encoded in 7 bits as shown in Table 5. Codes 0 through 37_8 in the ASCII code are control characters that control hardware but do not result in printed characters. Examples of this are TAB, CARRIAGE RETURN, etc.

The ASCII code for 7 is 0110111. Because 7 bits is so close to a byte, ASCII code is usually stored in memory one character per byte.

TABLE 5. ASCII CODE.

Meaning	Binary	Octal	Hex	Meaning	Binary	Octal	Hex
Space	010 0000	040	20	F	100 0110	106	46
!	010 0001	041	21	G	100 0111	107	47
"	010 0010	042	22	H	100 1000	110	48
#	010 0011	043	23	I	100 1001	111	49
$	010 0100	044	24	J	100 1010	112	4A
%	010 0101	045	25	K	100 1011	113	4B
&	010 0010	406	26	L	100 1100	114	4C
'	010 0111	047	27	M	100 1101	115	4D
(	010 1000	050	28	N	100 1110	116	4E
)	010 1001	051	29	O	100 1111	117	4F
*	010 1010	052	2A	P	101 0000	120	50
+	010 1011	053	2B	Q	101 0001	121	51
,	010 1100	054	2C	R	101 0010	122	52
-	010 1101	055	2D	S	101 0011	123	53
.	010 1110	056	2E	T	101 0100	124	54
/	010 1111	057	2F	U	101 0101	125	55
0	011 0000	060	30	V	101 0110	126	56
1	011 0001	061	31	W	101 0111	127	57
2	011 0010	062	32	X	101 1000	130	58
3	011 0011	063	33	Y	101 1001	131	59
4	011 0100	064	34	Z	101 1010	132	5A
5	011 0101	065	35	[	101 1011	133	5B
6	011 0110	066	36	\	101 1100	134	5C
7	011 0111	067	37	]	101 1101	135	5D
8	011 1000	070	38	^	101 1110	136	5E
9	011 1001	071	39	—	101 1111	137	5F
:	011 1010	072	3A	`	110 0000	140	60
;	011 1011	073	3B	a	110 0001	141	61
<	011 1100	074	3C	b	110 0010	142	62
=	011 1101	075	3D	c	110 0011	143	63
>	011 1110	076	3E	d	110 0100	144	64
?	011 1111	077	3F	e	110 0101	145	65
@	100 000	100	40	f	110 0110	146	66
A	100 0001	101	41	g	110 0111	147	67
B	100 0010	102	42	h	110 1000	150	68
C	100 0011	103	43	i	110 1001	151	69
D	100 0100	104	44	j	110 1010	152	6A
E	100 0101	105	45	k	110 1011	153	6B
				l	110 1100	154	6C
				m	110 1101	155	6D

ASCII numbers cannot be added together with a simple result; however, it is quite simple to convert ASCII numbers to binary by simply lopping off the upper bits.

OPERATION CODES

There is another kind of code used by the computer to control its operations, called operation codes (op-codes) or machine-language instructions. If a problem requires that two numbers be added, for instance, the computer must be instructed to perform the addition in the language of digital op-codes.

There is no standard for op-codes; they vary with each type of computer. For almost every computer, there will be some code that accomplishes addition; and 69_H is one op-code used for addition in the KIM-1 microcomputer which will be used in the lab.

A computer can accomplish complex tasks, but only if it receives instructions every step of the way. The op-codes for a computer define the only operations it can perform; and these operations are quite simple. To do something complex, the computer must be told to execute, in order, many op-codes which are carefully chosen to get the job done. This sequence of op-codes is called a program. One of the most difficult tasks related to computers is writing programs (programming); that is, figuring out the best op-codes to use to accomplish a given job.

EXERCISES

1. Write the type of output signal (analog or digital) that one would expect from circuits that can detect the following:
   a. Temperature
   b. Fire
   c. Burglar
   d. Weight
   e. The presence of a car
   f. Sunlight
2. Write the logical equivalent of the following voltages, using TTL definitions:
   a. 4.3 V
   b. 1.5 V
   c. 0.5 V
   d. 10.0 V
   e. 1.0 V
3. Find the hexadecimal and octal equivalents of the following:
   a. 101110101_2
   b. 101_{10}
   c. 110111111_2
   d. 39_{10}
4. Find the decimal equivalent of the following:
   a. 101101_2
   b. 37_8
   c. $3A_H$
   d. 10000000_2
   e. 77_8
   f. EE_H

5. Find the binary equivalent of the following:
   a. 31_8
   b. FF_H
   c. $1{,}000_{10}$
   d. 147_8
   e. ABC_H
   f. 255_{10}
6. Convert the following to binary and add; then, convert the result back to decimal and check, using decimal addition:
   a. 7 + 3
   b. 11 + 14
   c. 101 + 273
7. Write the following in ASCII code: CONSERVE ENERGY.

LABORATORY MATERIALS

Microcomputer (Commodore KIM-1).
Power supplies:
    5 volts at 1 A (TERC PS-005).
    12 volts at 100 mA (TERC PS-012).
Voltmeter.
Cassette tape recorder (Sanyo ST-45).
Software on cassette tape.
Connections to KIM output ports, power, and tape recorder.
Breadboarding system (TERC KIM-100).
500 ohm potentiometer.
NPN transistor (2N3392 or equivalent).
Resistors:
    27 ohm ¼ watt.
    1 k ohm ¼ watt.
Speaker, 8 ohm, 2" diameter.

LABORATORY PROCEDURES

INTRODUCTION

The laboratory objective in this module is to familiarize the student with the operation of a typical, small microcomputer – the KIM-1. The specific tasks are to measure and observe the various electrical and logical codes it uses.

Each major step is numbered and the number is followed by a major instruction. The paragraph(s) that follow the instruction explain how to accomplish the instruction, and often include important information, more detailed instructions and safety precautions needed to follow the instruction correctly. ALWAYS READ THE ENTIRE PARAGRAPH BEFORE TRYING TO FOLLOW THE NUMBERED INSTRUCTION.

LABORATORY 1: LOGIC VOLTAGES.

1. Make the connections to the KIM-1 as shown in Figure 9.

   All connections to the KIM are made through the KIMBOARD. ALWAYS CONNECT GROUNDS FIRST. Run a wire from the ground connector on the KIMBOARD (it has the legend (GND") to the minus and ground terminals on the +5 V power supply. If more than one power supply is used, interconnect all their grounds.

   Next make sure the power supplies are off. Then connect the 5 V supply to the +5 V terminal on the KIMBOARD. Connect the +12 V supply to the +12 V terminal. Plug the KIM-1 into the KIMBOARD.

   Finally, connect the interface board to the KIMBOARD, using the 20-conductor pink ribbon cable. Be sure to

insert the two white connectors the same side up. Recheck all connections very carefully.

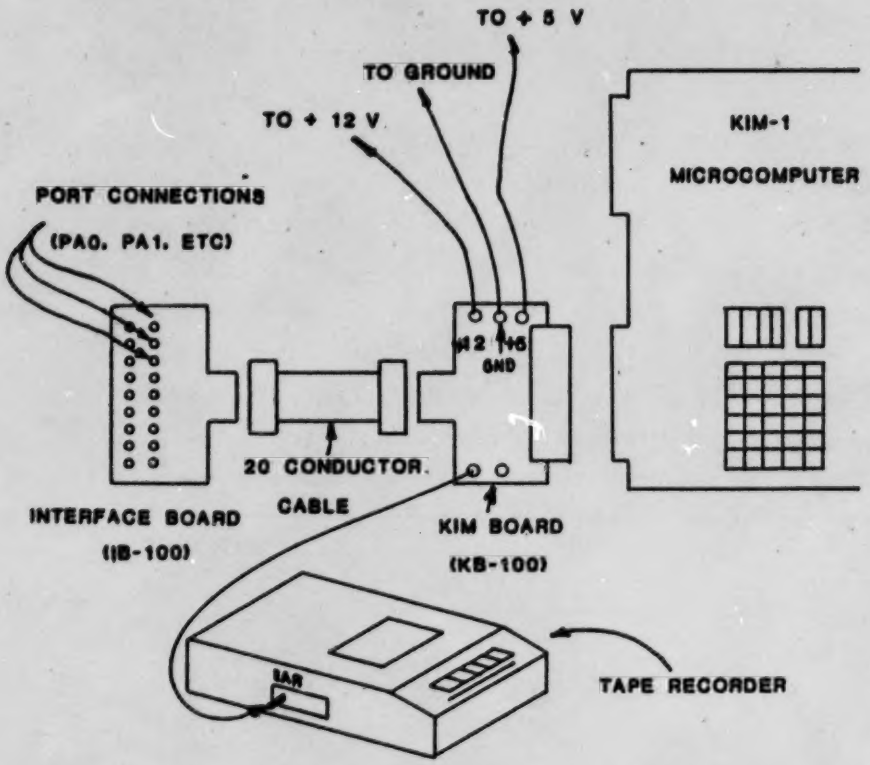


Figure 9. KIM-1 Connections.

2. Turn on and reset the KIM. Apply 5 V power to the KIM and press the reset button (marked "RS" on the keyboard). The 6-digit display should light; if it does not, quickly remove the power and get help.
3. Wire the test circuit shown in Figure 10. For the first experiment, apply various voltages and determine how the KIM interprets them. To do this, a variable voltage source is required; therefore, the 500Ω potentiometer, wired as shown in Figure 10, is used. As its knob is turned, the voltage on its center tap changes continuously

from 0 V to 5 V. Connect a voltmeter to this tap to measure the voltage.

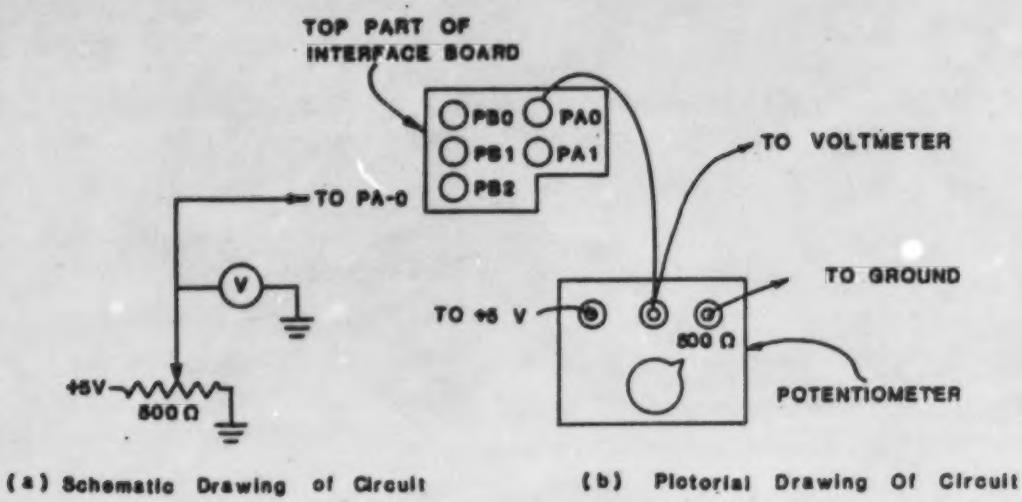
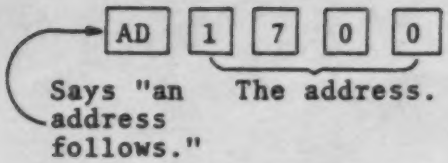


Figure 10. Circuit for Testing Input Voltage Levels.

The variable voltage is applied to one of the KIM inputs, called PA0. The KIM can read PA0 and display the result on its display.

4. Measure the transition voltage for PA0. The KIM has many internal memory cells (called addresses), each of which can store a byte of information. Each cell has a 2-byte address. The computer reads PA0 and displays the result when it is set to display the contents of address 1700. The KIM display shows 6 hexadecimal numbers: The left four numbers are always an address and the right pair shows the contents of that address. To find out what is in the address 1700, press the following five buttons on the KIM:



The left four digits should display 1700_H (the address); and the right four should display either FE_H or FF_H (indicating the contents of 1700_H). The display is FE if PA0 is a logical zero and FF if PA0 is logical one. Turn the 500 Ω potentiometer so that the voltmeter indicates that 0 V is being applied to the PA0. The display should indicate FE. Slowly increase the voltage until the display just turns to FF. Record the voltage in Data Table 1 (PA0 Transition Voltage). Repeat this twice. Now apply 5 V and slowly decrease the voltage until the display just changes from FF to FE. Record the voltage. Repeat this measurement twice. These measurements give six values of the threshold voltage PA0. Discard any that are far out of line and average the remaining measurements and record the results in Data Table 1.

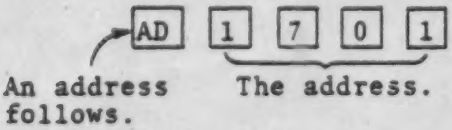
5. Repeat the measurements for the other PA lines. There are a total of 8 PA lines labeled PA0, PA1 ... PA7. These correspond to the 8 bits in the byte at address 1700_H. When no connection is made to any one of these lines, the computer reads that line as logical one. With no connections to any of the PA lines, the computer reads them as 11111111. It converts this to hexadecimal for the display. The hexadecimal equivalent of 11111111_2 is FF_H and this is displayed. If PA1 is at logical zero, the computer reads 1111 1101 and converts this to FD_H.

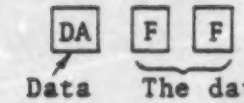
The kind of display used cannot display capital D since it would look like zero. The KIM uses the lower case, which looks like d . Likewise, B is displayed as b , and should not be confused with the number 6, which looks like 6 . See below for an expanded view:

Connect the test circuit in Figure 10 to PA1. The display should switch between FF and FD, depending on the logical value of the input at PA1. Repeat the measurements in Step 4 and use this to find the threshold voltage for PA1. One voltage reading is sufficient.

Repeat these measurements for all the remaining ports, PA2 to PA7. In each case, the display alternates between FF and some other number. Record the readings in Data Table 1 (Other PA Transition Voltages) and explain these numbers. These measurements give the transition voltage for each port. Are they all the same? Would an indeterminant region be needed for these?

6. Measure the KIM's output voltages. In this step the KIM is used to generate logical outputs and the voltages of these will be measured. First, disconnect the 500Ω potentiometer circuit in Figure 10. Then make PA generate logical outputs by placing FF into address 1701_H. This can be done by pressing the following keys:





From now on, do not press reset because it changes the contents of 1701_H. (If it is pressed, FF must again be loaded into 1701_H.) Now generate logical ones on all the PA lines by placing FF into 1700_H. Do this by pressing the following:

AD 1 7 0 0 DA F F

Use the VOM to measure and record the voltage on each of the eight PA lines. Record in Data Table 1 (KIM Output Voltages). Generate logical zeros on the PA lines by pressing DA 0 0 . Again, read and record the voltage on each of the eight PA lines. What is the range of voltages the KIM generates for the two logical states?

LABORATORY 2: THE SOUND SYNTHESIZER.

1. Build the audio amplifier in Figure 11.

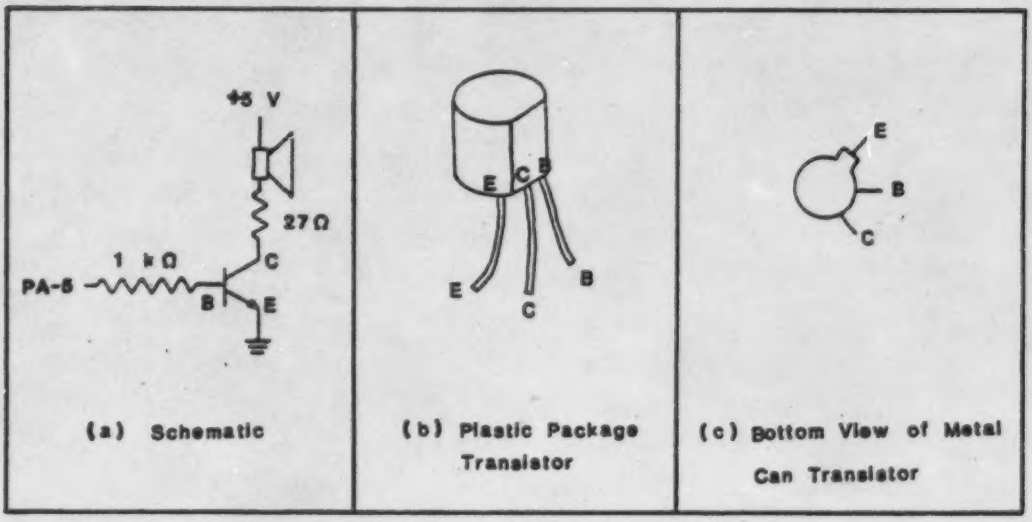


Figure 11. Schematic of an Audio Amplifier and Transistors.

The next step is to enter and alter some programs that can make quick changes in the output voltages. The easiest way to detect these voltages is to convert them into sound. If a voltage regularly alternates between logical zero and one, it can produce a steady tone. A small amplifier is needed to hear the sound. Figure 11a shows the required circuit. The central element is an NPN transistor. The three leads on the transistor are called the emitter (E), base (B), and collector (C). Figure 11b shows how to identify these for two common transistor packages. Connect the circuit with the power off.

2. Enter the program. Because this step makes extensive use of the KIM, learn these rules about its operation:
   a. The [AD] and [DA] buttons are like a switch; the computer remembers which of the two was last pressed.
   b. If [AD] was last pressed, then any number pressed from 0 to F goes into the address. Any number of keys can be pressed, but the last four always give the address.
   c. If [DA] was pressed last, then any number pressed becomes data and is placed into the address being displayed. This alters the contents of memory.
   d. The [+] button adds 1 to the address. This button makes it easy to change the contents of several addresses in a row.
   e. The [GO] button starts a program to executing the op-code that is displayed on the right and located at the address on the left of the display.

   Hereafter, the buttons to be pressed will not usually be stated explicitly in the lab procedures; so, use these five rules to translate the lab procedures into key strokes. Enter the following data into the first six memory addresses, starting at 0000:

   [EE] [00] [17] [4C] [00] [00]

   Go back to 0000 and check these six addresses.

3. Execute the program. Place FF in 1701. Slide the black switch on the keyboard OFF, away from the ON legend. Now start the program at 0000 by pressing the following:

   [AD] [0] [0] [0] [0] [GO]

   Says "start the computer" at address 0000.

The display should blank out and the speaker should generate a high-pitch sound. What happens when the amplifier signal is obtained from PA outputs other than PA5? Record the results for all other ports in Data Table 2 (Program Execution). To stop the computer, press [RS] (reset). Reset always changes the contents of 1701_H. To avoid this, place 00_H into $17FA_H$ and $1C_H$ into $17FB_H$. Now the [ST] (stop) button will usually stop any program without altering 1701_H.

4. Explore the effect of other op-codes. The mechanism used by the computer to make the quickly changing outputs has not been discussed. That will be covered in another module. Here, it is sufficient to verify that the codes entered cause the sound output. Try altering the codes in any of the first six addresses and then starting the program again at 0000. Note that reset puts 00 into 1701_H; so, to get any result, FF will have to be reloaded into 1701_H each time the computer is reset. Does altering data in any other address affect the sound? Record the observations in Data Table 2 (Other Op-codes).

5. Load and run other programs. Other longer programs are recorded on cassette tape. The following procedure should always be used to read taped programs into the KIM. Each tape can have many programs on it. To tell them apart, each can have a 2-digit hexadecimal number called the ID (identification).

   a. Place 00 into address $00F1_H$.
   b. Place the ID number in address $17F9_H$.
   c. Start executing at address 1873_H.
   d. Advance the tape to near the beginning point of the program, if known. Connect the tape output marked EAR into the computer tape input. Turn the volume to maximum and turn the tone control to full treble. Check that 12 V is applied to the KIM.

e. Start the tape reading. When four zeros appear, the tape has been successfully read. If four F's appear, or if nothing appears after a few minutes, the tape was not read properly. Try again before asking for help.

Read in the program using 01 for the ID. The starting address for the program is 0000. With the speaker circuit connected to PA5, run the program and describe the results in Step 5 of Data Table 2 (The Taped Program).

DATA TABLE 1: LOGIC VOLTAGES.

Step 4: PA0 TRANSITION VOLTAGE

Measurement	Voltage
1	
2	
3	
4	
5	
6	

Average voltage: _____.

Data Table 1. Continued.

Step 5. OTHER PA TRANSITION VOLTAGES.

PA Line	Voltage	Display Seen When Line Is Low	Binary Equivalent
1			
2			
3			
4			
5			
6			
7			

Explain the results in Columns 2 and 3: _____

_____

_____

Indeterminant region for PA lines: _____

Step 6: KIM OUTPUT VOLTAGES

	Measured Voltage	
PA Line	Logical 0	Logical 1
0		
1		
2		
3		
4		
5		
6		
7		

Range of logical zero voltages: _____

Range of logical one voltages: _____

DATA TABLE 2: THE SOUND SYNTHESIZER.

Step 3: PROGRAM EXECUTION

Description of sounds:

PA Line	Description
0	
1	
2	
3	
4	
5	
6	
7	

Step 4: OTHER OP-CODES

Address	Op-Code Used	Result
0000		
0001		
0002		
0003		
0004		
0005		

Step 5: THE TAPED PROGRAM

Describe the output.

REFERENCES

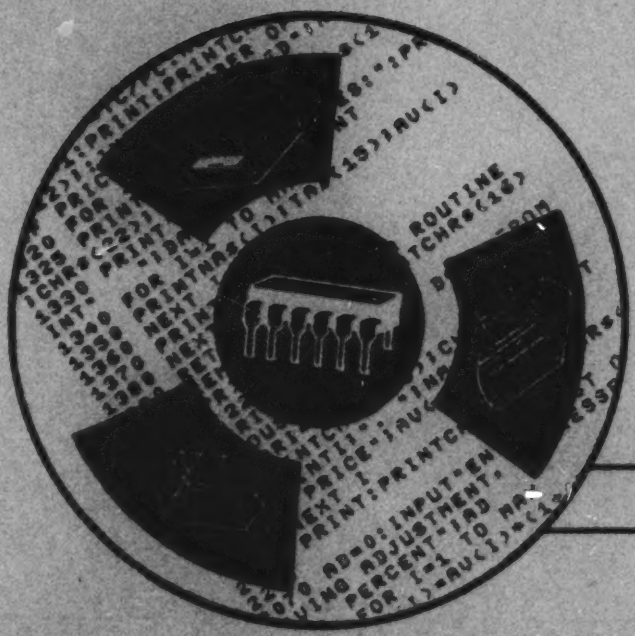
Foster, Caxton. Microcomputer Programming: The 6502. Addison-Wesley.

KIM-1 User's Manual. Norristown, PA: MOS Technology.

Tinker, Robert F. Microcomputers. TERC, 1978.

ENERGY TECHNOLOGY

CONSERVATION AND USE



MICROCOMPUTER OPERATIONS

MODULE MO-02

MICROCOMPUTER ARCHITECTURE

CENTER FOR OCCUPATIONAL RESEARCH AND DEVELOPMENT

INTRODUCTION

"Architecture" refers to the "building blocks" or physical components within a computer that perform its operation. To understand how a digitally-coded input is "processed" within the computer to produce a desired output, the major components, called "hardware," must be examined.

The "brain" of a microcomputer is a small chip, or integrated circuit (IC), such as Intel's 8080, Zilog's Z-80, Motorola's 6800 or MOSTECH's 6502. Each type of chip has its own "instruction set," which is a list of things it can do - such as "move information stored in memory location 30 to the accumulator (temporary storage)," "increase number stored in the accumulator by 1," or "subtract number stored in memory 30 from number stored in accumulator," etc. The chip is the heart of a unit called the Central Processing Unit (CPU), which controls the flow of data (information) and performs all computations. The chip routes the signals along lines (called "buses") from memory locations that are distinguished from each other by their "address."

Just as an automobile engine runs only when the valves open and close at the right time, relative to the piston's operation, so, too, the operation of a microprocessor is dependent upon a "clock" which instructs the various parts of a computer. These clock pulses are routed to all components via a "control" bus.

The primary purpose of this module is to explain how computer hardware, via its architecture, performs the task assigned to it. By the end of the module, the student should be able to predict what the computer will do with instructions given it.

PREREQUISITES

The student should have completed Module MO-01 of Microcomputer Operations.

OBJECTIVES

Upon completion of this module, the student should be able to:

1. Explain the function and significance of major architectural features of a computer and the 6502 microprocessor, including the CPU, RAM, ROM, I/O Ports, Accumulator, index register, and program counter.
2. Examine and alter memory and CPU registers on a KIM microprocessor.
3. Single step through programs that use the op-codes in Table 2 and verify the effect on each step.
4. Define the following terms:
   a. Interface.
   b. Program.
   c. Algorithm.
   d. Programming.
   e. Central processing unit.
   f. Volatile.
   g. Input/output ports.
   h. System I/O.
   i. X index register.
   j. Bugs.
   k. Application I/O.
   l. Addresses.

m. Invalid address.
n. Operand.
o. Mnemonic.
p. Monitor.
q. Bus.
r. Control bus.
s. Address bus.
t. Data bus.
u. Accumulator.
v. CPU register.
w. Program counter.
x. Single-step mode.
y. Machine instructions.

5. Identify the following abbreviations:
a. CPU
b. RAM
c. ROM
d. RWM
e. I/O ports
f. PC
g. X register
h. OR
i. AND
j. XOR

A typical control application of a microcomputer is shown in Figure 1. To have a particular chemical reaction proceed in a desired way, the microcomputer in Figure 1 has been programmed to control the temperature of a reaction vessel. It can, for instance, slowly raise the temperature to a predetermined level and then let it drop quickly to a lower temperature. The microcomputer determines what the temperature is in the reaction vessel with a temperature sensor that generates an analog signal. Since the computer cannot read an analog signal directly, this signal is converted to a digital signal with an A/D convertor (ADC).

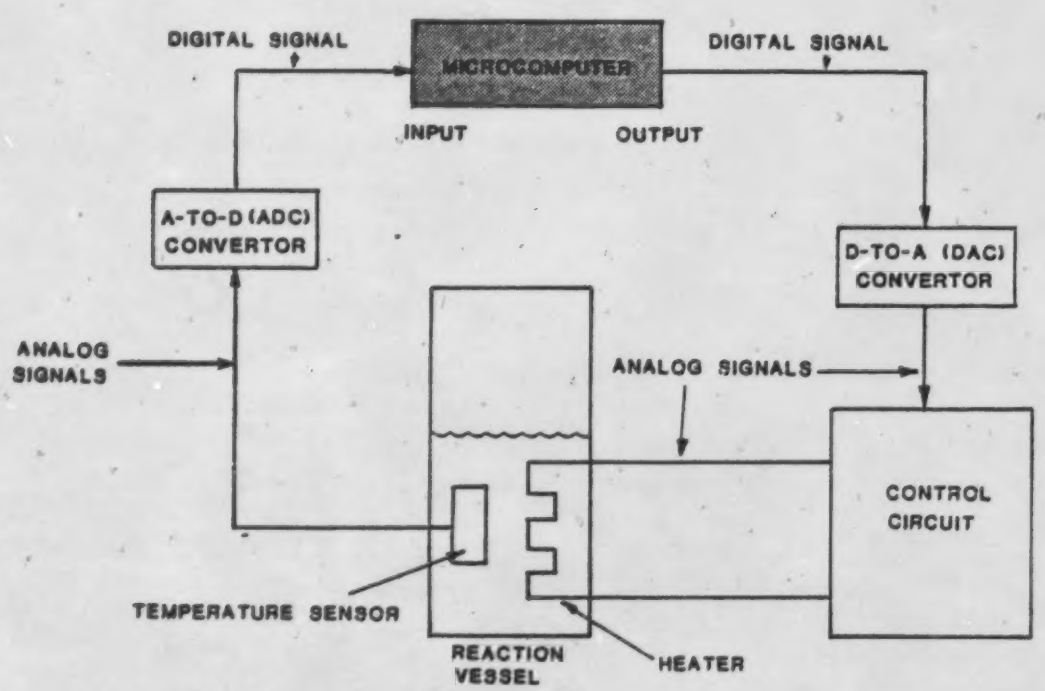


Figure 1. A Typical Computer Application.

On the output side, the computer controls the amount of power delivered to a heater. It cannot do this directly, because the heater requires more power than the computer can deliver and because the varying amount of power is inherently an analog signal. The computer must first convert its digital output to an analog level. This analog signal is then used to control the amount of power applied to the heater by an appropriate control circuit.

This example is typical of many applications of microcomputers. Since the computer is a digital device, it must have digital input and output signals. As a result, the computer itself normally can be represented as a black box with a certain number of digital input and output lines connected, as shown in Figure 2. In applications such as the one involving the temperature controller shown in Figure 1, where the computer's input/output digital signals are not appropriate, special circuits like the A/D and D/A convertors must surround the computer. The general term for the circuits that go between the computer and the real world is <u>interface circuits</u>. Many different kinds of interface circuits exist, each suited to adapt the computer to one or another application.

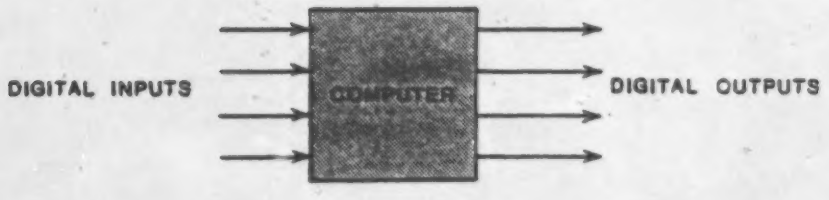


Figure 2. A Computer as a Black Box.

The connection between the inputs and outputs of a computer are determined by the program, which is a sequence of of simple instructions that the computer executes. For example, the microcomputer portion of the temperature controller in Figure 1 could be programmed to read the input signals and determine the actual temperature of the reaction vessel, then to compare this temperature to the desired temperature and make a suitable correction to the amount of power being applied to the heater.

A step-by-step procedure for solving a problem is called an algorithm. Many different algorithms are possible. The simplest might be something like this: "if the temperatue is too low, increase the heat; if it is too high, decrease the heat." Or it might be much more complex and consider factors such as the length of time since the heat was last changed, the amount of material in the reaction vessel, the outside temperature, and so forth. Whatever the algorithm is, a programmer can translate it into instructions the machine can understand. This is called programming.

A computer is a general-purpose device which can be applied to many practical problems. Two things must be done to apply a computer to a particular problem: it must be interfaced and programmed. Interfacing adapts the computer to the particular real-world hardware and gives the computer the ability to test and control variables in a particular situation. The program takes the general, simple instructions of the computer and puts them in proper sequence to accomplish the specific task at hand. In the sections that follow, the student will see how the interface and programs can be made to work together. Even simple applications require a good understanding of the hardware interfacing and of the software program.

DIGITAL INTERFACES

The home security system illustrated in Figure 3 uses a variety of switches and light beams to detect whether or not a burglar is in the house. If the microcomputer "thinks" there is a burglar, it rings an alarm. This particular application requires digital inputs and outputs. Each sensor (digital device) tells whether or not it detects an intruder; either there is or there is not an intruder. Likewise, the output is digital; the computer either rings the bell or it doesn't. In this particular case, the logic required in the real world is compatible with the logic of the computer.

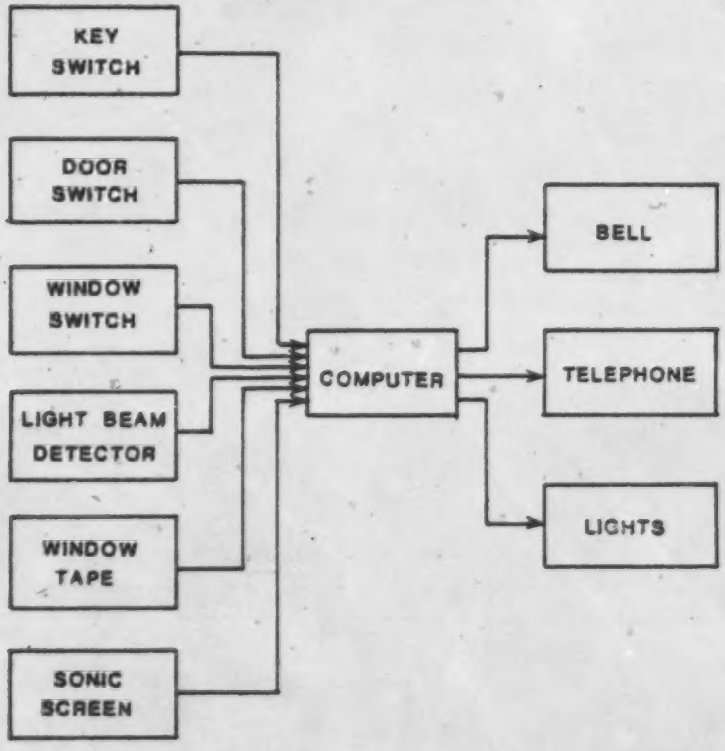


Figure 3. A Microcomputer Security System.

However, this does not mean that the sensors can be plugged directly into the computer. The sensors and the computer are not necessarily electrically compatible; sensors may not generate the exact voltages that are needed on the computer inputs and the computer output may not generate the kind of signal necessary to ring the bell. In this case, some interfacing is often required to match the electrical needs of the various components to the requirements of the computer.

ANALOG INTERFACES

Analog signals are needed in many applications. The usual way to produce an analog signal is shown in Figure 4. An 8-bit digital number is produced by the computer and a special circuit is used to convert this to an analog voltage. A typical approach is to have an increase of 1 in the digital number result in an increase of 0.04 V in the analog output. In other words, the number 00_H would result in 0 V output, the number 01_H would result in 0.04 V output and the number 02_H would result in 0.08 V output. A general equation for output voltage is as follows:

$$V = 0.04 \times N \qquad \text{Equation 1}$$

where:

V = The output voltage.

N = The applied binary number.

The largest possible 8-bit output is $1111\ 1111_2$ or FF_H (which is the decimal number 255). Using Equation 1, this converts to 10.2 V (0.04V/step x 255 steps = 10.2V).

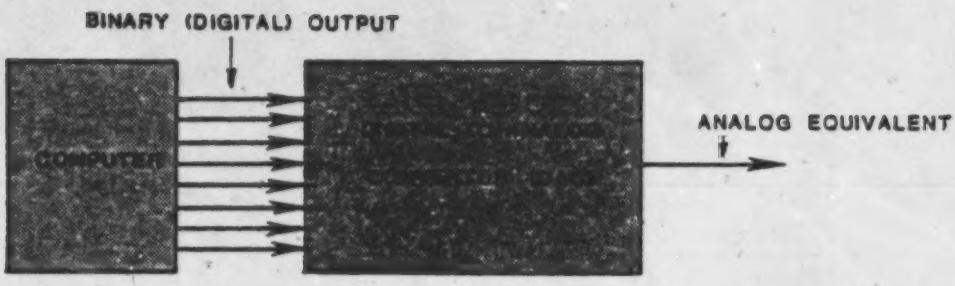


Figure 4. Digital-to-Analog Convertor.

The convertor in Figure 4 permits the computer to generate analog signals in the range between 0 to 10.2 V. The voltages are generated in steps of 40 mV (0.04V), which means that the computer cannot generate any arbitrary voltage, but it can get within 20 mV in any desired voltage. For instance, suppose the computer needed to generate exactly 6.175 V. Dividing this number by 0.04 V (the size of one step) yields the total number of steps required to generate this voltage. The result is the following:

$$\frac{6.175\ V}{0.04\ V} = 154.375 \text{ steps}$$

In other words, 154.375 steps of 40 mV each would be required to generate this voltage. Of course, the computer cannot generate a fraction of a step, but could come close by generating 154_{10} in binary, which would result in the following output voltage:

$$154 \times 0.04 = 6.16 \text{ V}$$

This is different from the desired voltage by only 15 mV.

EXAMPLE A: D/A CONVERSION.	
Given:	A 10-bit digital/analog convertor that generates 8 mV per step.
Find:	The maximum voltage and the digital output required to generate a voltage close to 6.175 V.
Solution:	The maximum value for a 10-bit D/A convertor would be $11\ 1111\ 1111_2$, or $1{,}023_{10}$. If each step were 8 mV, then 1,023 of them would be 8.184 V. This is the maximum voltage a convertor could generate. A voltage of 6.175 represents the following number of 8 mV steps: $\frac{6.175}{0.008} = 771.875$ steps. The best the computer could do would be to output the digital integer, 772. This would result in an output voltage of $772 \times 0.008 = 6.176$ V. In this case, there would only be 1 mV error.

Special analog-to-digital convertor circuits also exist (as shown in the block diagram in Figure 5). When a voltage within a specified range is applied, these can generate the

corresponding digital number. In this way, the computer can obtain the value of analog voltages that are applied by reading the digital output of the analog-to-digital convertor. Again, there is some inaccuracy because of the digital nature of the final result.

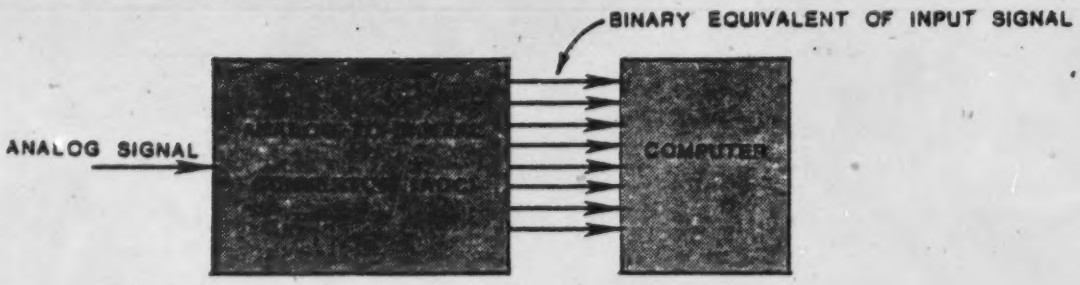


Figure 5. Analog-to-Digital Convertor.

EXAMPLE B: A/D CONVERSION	
Given:	An 8-bit A/D convertor with 0.04 mV per step.
Find:	The digital output generated when 1.75 V is applied.
Solution:	First calculate the number of 0.04 V steps in 1.75 V.
	(number of steps) = $\frac{1.75}{0.04}$ = 43.75 steps
	The ADC should generate the next nearest integer, 44. Some ADCs will ignore the fractional part and generate 43. Either answer is correct.

COMPUTER COMPONENTS

The primary components within a typical microcomputer are shown in Figure 6. The central processing unit (CPU) controls the entire system and also performs the arithmetic and logic operations required. Two forms of memory are shown: random-access-memory (RAM) and read-only-memory (ROM). Ramdom-access-memory is memory that can be both read from and written into by the CPU. For this reason, a more accurate name for this type of memory would be read-write memory (RWM). ROM memory only can be read from, but not altered by, the CPU under normal circumstances.

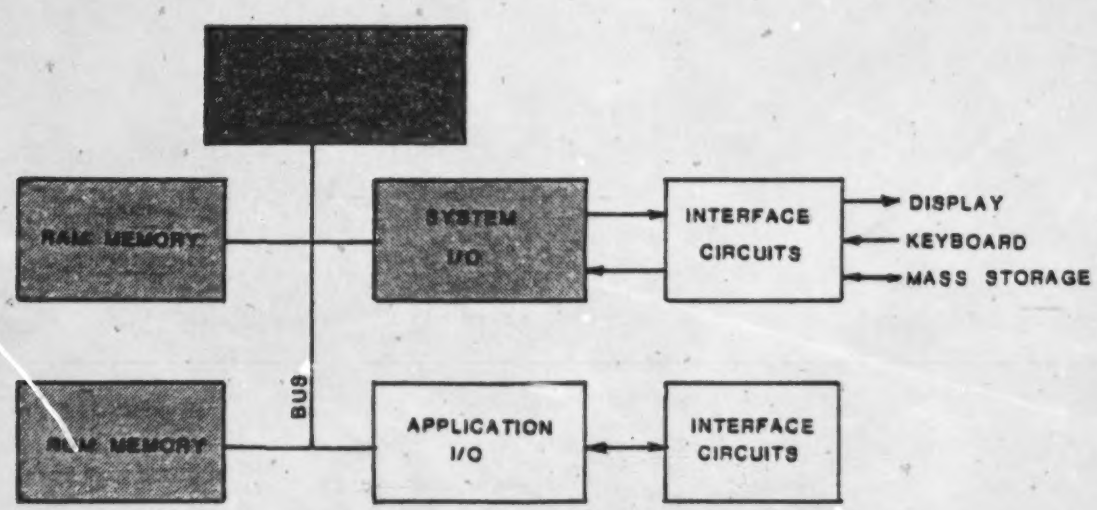


Figure 6. Primary Components within a Typical Microcomputer.

The RAM memory universally found in small computers is volatile; that is, it loses its contents when power is removed from the system. To have programs and data available to the computer when it is first powered, a non-volatile memory is required. ROM memory is non-volatile and is available as soon

as the computer is powered up – which is the reason most microcomputer systems have ROM memory.

When the computer communicates with the outside world, it uses input/output ports (I/O ports). Two kinds of I/O ports are shown in Figure 6: One type of I/O port is called system I/O, because it has circuits permanently connected to it that are necessary for the operation of the computer system. The other type of I/O is called application I/O. This latter I/O port is not just dedicated to use by the computer for its own purposes; it is available for specific applications of the computer. For instance, switches, speakers and A/D and D/A convertors could be connected to the application I/O for particular applications. (Not every computer will have all these components.)

ADDRESSES

The computer system memory is organized by addresses. Each address can be thought of as a "cubby hole," where data can be stored or from which data can be retrieved. If the CPU is an 8-bit CPU, then exactly 8 bits of information can be stored at or retrieved from each address. Similarly, 12- and 16-bit CPUs store 12 and 16 bits at each address.

CPUs that are 8-bit almost always provide 16 bits of address information (in two groups of 8 bits each). In this way, they can address 2^{16}, or 64K (65536), different addresses rather than just 2^{8}, or 256 addresses. This does not mean, however, that every microcomputer system can use that many different addresses; it simply is the limit on the total number of addresses that could be used in such a computer system. The memory used in most microcomputer systems is much less than 64 kilobytes (usually 2k to 16k), and as a result, many addresses are usually invalid or inoperative. It is the responsibility of the programmer to be sure that addresses used in a program do correspond to valid memory locations.

KIM – A SPECIFIC EXAMPLE

Figure 7 shows some of the detail of the KIM-1 microcomputer architecture. One kilobyte of RAM memory resides at addresses 0 through $3FF_H$ (1024_{10}) and two kilobytes of ROM memory reside at 1800_H (6144_{10}) through $1FFF_H$ (8192_{10}). System I/O ports start at address 1740_H (5952_{10}) and application I/O ports start at 1700_H (5888_{10}).

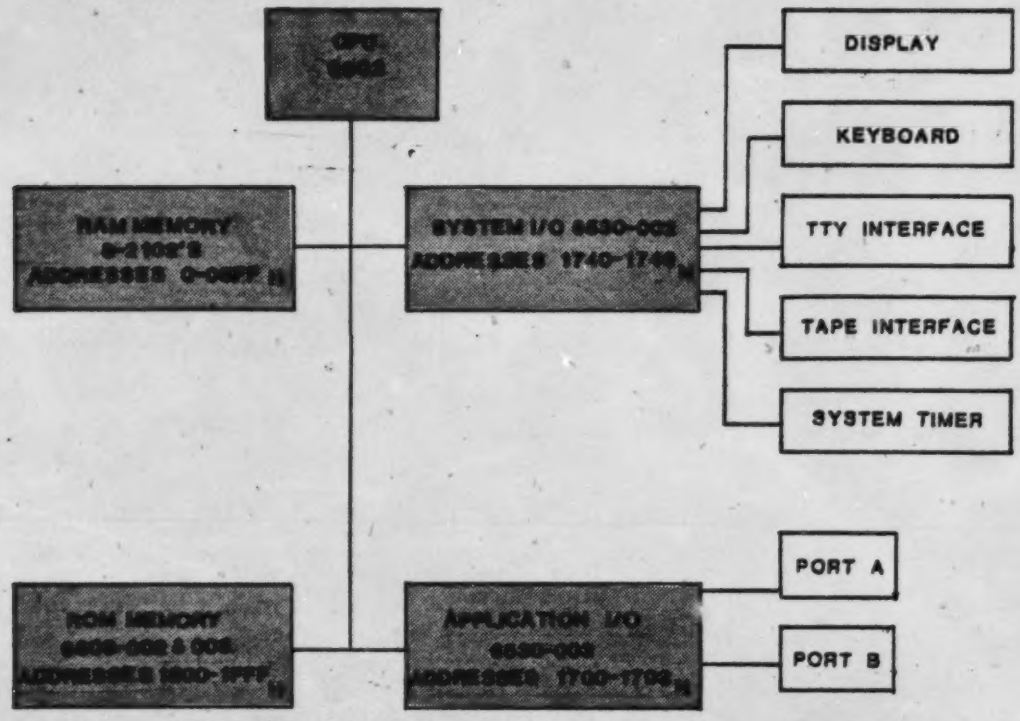


Figure 7. The KIM-1 Architecture.

When the KIM is first turned on, special circuits cause it to start executing programs which are stored in ROM memory. These programs are responsible for using the system I/O to light the displays and examine the keyboard for depressed keys.

As discussed in the previous module, various keys on the keyboard cause certain important functions to be performed, such as displaying and altering memory, and loading, executing and recording programs. All of these operations are peformed by the CPU in executing programs stored in ROM memory. Without the ROM memory, nothing would happen in the KIM-1 system - even with the power applied, there would be no way of entering programs, no way of executing them and no way of examining the contents of various locations. The set of programs stored in ROM memory, which permit all this to happen, are collectively known as the monitor.

Although not used in this module, the KIM has monitor programs and interface hardware that can use a teletypewriter (TTY) which is useful for providing typed output of programs and data. In the KIM, an output looks just like another memory bank; that is, the CPU can store and retrieve information from an address which is known as the output port. The only difference between this address and the RAM memory address is that, at an output port address, data can be read by the outside world. There is no direct way of reading and measuring the data in RAM memory. Similarly, input ports are like ROM memory; because they are inputs, the CPU cannot alter the contents. Instead, the contents are set by the voltages applied from the outside world. However, to the computer, the input appears to be just another address which can be read at any time...just like ROM memory. The only difference is that ROM memory never changes, whereas an input port changes if the applied voltages change.

THE BUS

The CPU controls all operations within the computer through a set of wires known as the bus. Within the bus there are three groupings known as the control bus, the address bus, and the data bus. The CPU controls the flow of information within the

computer through two basic operations: reading and writing. When it needs to read information, it generates signals from the control bus which say, in effect, "give the CPU information now." At the same time, it places the address, from which the data should be read, on 16 address lines. Then it waits for the external circuitry to retrieve the information and put it on the 8 lines of the data bus. This operation is illustrated in Figure 8a.

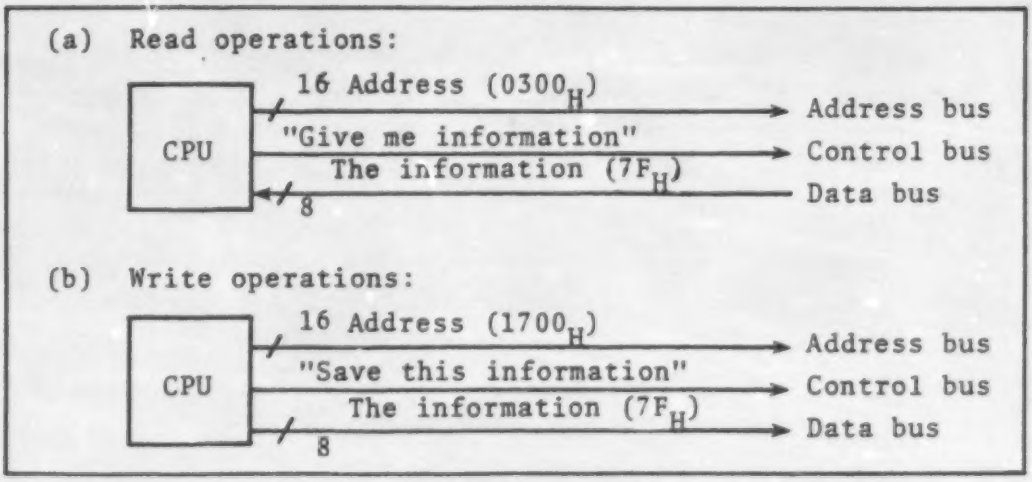


Figure 8. Bus Signals for Read-and-Write Operations.

When the CPU must store information somewhere, it generates a signal on the control bus which says, in effect, "write or save this information." At the same time, it puts the address (where the information should be stored) on the address bus and places the data (the information to be stored) on the 8 lines of the data bus, as shown in Figure 8b.

CPU ARCHITECTURE

To understand how the CPU accomplishes its dual tasks of control and computation, the CPU must be examined. In the following sections of this module, some of the components inside the KIM's CPU are detailed and the operation of these components are illustrated. Although the KIM is used as a specific example, the major architectural features are common to most microcomputers.

THE ACCUMULATOR

The CPU contains several special memories, called registers. The most important of these is the accumulator, or the A register.

The CPU cannot move data directly from one address to another. If, for some reason, data must be moved from address 0300_H to address 1700_H, this would be accomplished in two steps. The first step consists of moving the data from 0300_H to the accumulator in the CPU. Then, in the second step, the data is moved from the accumulator to 1700_H. In this example, the accumulator is used as a convenient intermediate place to put data while it is being moved. In other situations, the accumulator is used to store one of two numbers to be added, and to store the result of the addition.

The KIM is an 8-bit machine; its accumulator holds 8 bits of data.

THE PROGRAM COUNTER

A second important CPU register is the program counter, or PC. The PC always holds the address of the next instruction to be executed. Normally, instructions follow one another in sequence. Instructions can be 1, 2 or 3 bytes long. There are circuits in the CPU that examine instructions and determine the number of bytes in each one. This number is automatically added to the PC to get the address of the next instruction.

However, instructions do not always follow in sequence. In these instances, the PC must be altered to hold the address of the instruction that is not in sequence. This is accomplished with JUMP and BRANCH instructions.

The PC holds an address. Since addresses are 16 bits long, the PC is a 16-bit register.

THE INDEX REGISTER

The final CPU register discussed in this module is the X index register, or X register. Like the accumulator, this register can be used for temporary storage, but it also has other important functions. Because it is a data register in the KIM, it also is 8 bits long.

EXAMINING CPU REGISTERS

Programs seldom work correctly the first time. To help find the errors, or "bugs" as they are called, the KIM has a single-step mode. In the single-step mode, the execution sequence of the CPU is held in suspension after each instruction by a WAIT signal.

When single-stepping through a program, it is sometimes useful to be able to examine the contents of the three CPU registers discussed. The registers do not have addresses, so the usual method of examining the contents of addresses does not work. The KIM monitor solves this problem in single-stepping by displaying the PC, storing the accumulator in address $00F3_H$, and storing the X register in address $00F5_H$. This makes it possible to examine the progress of a program between steps. To determine the contents of the accumulator, simply examine the contents of address $00F3_H$ – they will be the same.

ASSEMBLY AND MACHINE CODE

The instructions understood by the CPU are called machine instructions, which consist of a 1-byte op-code followed by zero and one or two bytes of data called the operand. Each op-code has a fixed number of operands that must be supplied.

It is quite difficult to interpret the machine code directly. Programmers usually use short abbreviations for the op-codes, called mnemonics. For example, an instruction loads the accumulator with data at some address in memory. The machine code for this is $1010\ \ 1101_2$, or AD_H; but the mnemonic is LDA (LoaD Accumulator), which is much easier to remember. Table 2 shows a partial listing of the "instruction set" of the MOSTECH 6502 chip used by the KIM-1 microprocessor's CPU.

It is also convenient to use numerical addresses. Usually, data stored at some location have some particular meaning and it is easier to give the address a name that is related to the meaning. For instance, port A on the KIM is at address

1700_H. It is convenient to refer to that address as PORTA, or PA. In this context, PA is called a label, or a symbolic address.

A program written with mnemonics and symbolic addresses is called an assembly language program. The KIM will not understand an assembly language program; such programs are simply conveniences for programmers. Any assembly language program must be converted into machine language before it is entered into a computer and executed. This conversion process is called assembly. Computers can be programmed to perform assembly, but this module will show how it is done "by hand."

A SIMPLE PROGRAM

Table 1 illustrates a short program written in both assembly code and machine code. The assembly code version of that program will be examined first.

TABLE 1. A SHORT PROGRAM.

ASSEMBLY CODE			MACHINE CODE (IN HEX)				
Label	Mnemonic	Operand	Address	Op-Code	Operands		Comment
START:	INX		0010	E8			Adds 1 to X register
	STX	PA	0011	8E	00	17	Stores X in port A
	JMP	START	0014	4C	10	00	Return to start

The first mnemonic is INX, the abbreviation for "INcrement the X index register by one"; that is, add one to the X register. No operand is required. After doing this, the CPU will STX, which is the abbreviation for "STore the contents of the X index register at the address given by the operand." The operand is PA, so the instruction transfers the contents of the X register to the output port A. This operand is an address, so it requires 16 bits or 2 bytes.

The final instruction is JMP, an abbreviation for "JUMP (go directly) to the address given by the operand." The effect of this instruction is to load the operand into the PC. This is a case where the next instruction is not in sequence. The operand is START, which is the symbolic address given to the first instruction. As a result, the following instruction executed will be INX again.

This program, then, is an endless loop. It adds one to whatever is in X, moves the result to the output port A, and then repeats the process endlessly.

Now the assembled machine code version of this program is examined. The first problem is to determine the starting address for the program. In this example, 0010_H was chosen arbitrarily.

Op-codes for the mnemonics used in this module are shown in Table 2. This table shows that the op-code for INX is $E8_H$, so this will be stored at address 0010_H. No operand is needed for INX, so the next instruction starts at 0011_H. In this address, the op-code for STX ($8E_H$) is stored. STX needs a 2-byte operand, the address for PA. The address is 1700_H, but it is entered with its bytes reversed; 00 in 0012_H and 17 in 0013_H. The KIM's CPU always requires addresses in this reverse byte order, which is another reason that assembly language is easier to read than machine codes.

TABLE 2. TYPICAL IIM INSTRUCTIONS.

MNEMONIC	MODE	OP-CODE (IN HEX)	MEANING
ADC	Absolute	6D	Add data at the address given by the operand to the accumulator with carry.
AND	Absolute	2D	AND the accumulator data with data at the address given by the operand.
ASL	Accumulator	0A	Arithmetic shift left of the accumulator. Shifts 0 into bit 0, bit 7 into carry.
BEQ	Relative	F0	Branch if equal to zero.
CLC	Implied	18	Clears the carry.
DEX	Implied	CA	Decrease the data in the X index register by one (subtract one).
INC	Absolute	EE	Add 1 to data at the address given by the operand.
INX	Implied	E8	Add 1 to the contents of the X index register.
JMP	Absolute	4C	The next op-code is at the address given by the operand.
LDA	Immediate	A9	Load the accumulator with the operand.
	Absolute	AD	Load the accumulator with data at the address given by the operand.
LDX	Absolute	AE	Load the X at the address given by the operand.
LSR	Accumulator	4A	Logical shift right of the accumulator. Shifts 0 into bit 7; bit 0 into carry.
STA	Absolute	8D	Store the accumulator data at an address given by the operand.
STX	Absolute	8E	Store the X register data at the address given by the operand.

The JMP instruction op-code is $4C_H$. The operand for this instruction is the address of the first instruction where the label START is. This address is 0010_H, which is entered in reverse order into addresses 15_H and 16_H.

EXAMPLE C: PROGRAM ASSEMBLY.

Given: The following assembly language code.

```
FIRST:  INC PA
        JMP FIRST
```

Find: The machine language version of this code. Start the code at 0300_H and use 1700_H for PA.

Solution:

ADDRESS	HEX. OP-CODE	OPERANDS
0300	EE	00 17
0303	4C	00 03

The two op-codes are found in Table 2. The operand for INC is 1700_H, but this is entered in reverse-byte order. The operand for JMP is the address for the label FIRST, 0300_H. This, too, is entered in reverse-byte order.

BRANCHING AND IMMEDIATE DATA

The program just discussed is quite fast. The output will increase every 9 microseconds. If a slower rate of increase is needed, one way to accomplish this is to have the computer waste time by counting a certain amount each time around the loop. Table 3 illustrates this.

TABLE 3. A SLOWER PROGRAM.

ASSEMBLY LANGUAGE		MACHINE CODE (IN HEX)			
Label	Mnemonic Operand	Address	Op Code (IN HEX)	Operands	Comment
START:	INX	0020	E8		Increment X
	STX PA	0021	8E	00 17	Output result
	LDA #8	0024	A9	08	Set count to 8
	STA COUNT	0026	8D	34 00	
CYCLE:	DEC COUNT	0029	CE	34 00	Subtract 1
	BEQ JUMP	002C	F0		Cycle back, if not zero
	CYCLE	002E	4C	29 00	
JUMP:	JMP START	0031	4C	20 00	If zero, start over

The program is easier to understand from its flow chart shown in Figure 9. This program is like the previous one, except that a new loop has been added. In this inner loop, 8 is stored in some address, called COUNT. Then one is subtracted from COUNT. If the result is not zero, the program loops back and subtracts one again. It keeps subtracting one until nothing is left in COUNT. It will take 8 times through the loop for COUNT to become zero. When it is finally out of this inner loop, the program starts over and increments X.

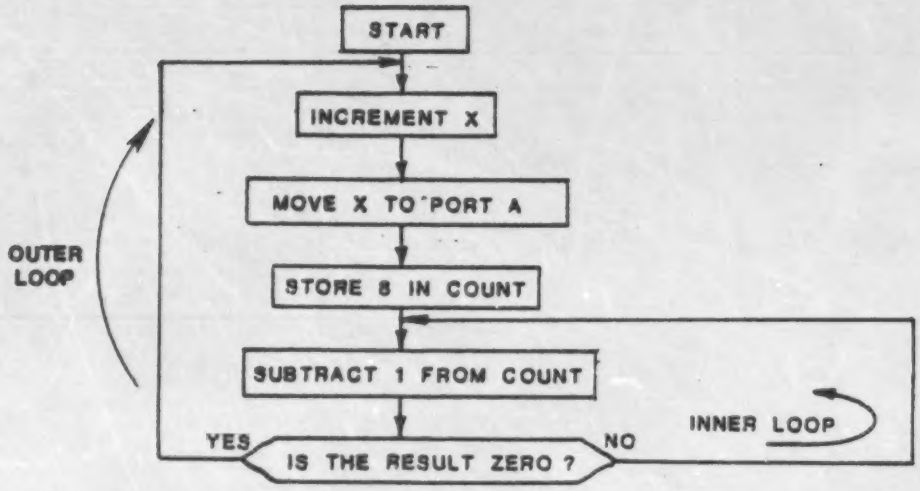


Figure 9. Flow Chart of the Program in Table 3.

The net effect of the inner loop is to waste time subtracting one from COUNT eight times. Each subtraction takes 12 microseconds, so this program wastes 96 microseconds. The number of subtractions is determined by the number initially placed into COUNT. This example shows 8, but larger numbers could be used. If FF_H were used, a delay of 3,060 microseconds would result.

In the assembly code shown in Table 3, the first two instructions are the same as the program shown in Table 1. The third instruction, LDA #8, illustrates a new concept. The effect of this instruction is to load the operand 8 into the accumulator. This is different from the instructions used so far, where the operand was an address of data. Here, the operand is the data.

Use of the operand is called immediate addressing mode, because the data immediately follows the op-code. The pound sign (#) is usually used to signal this mode and distinguish

it from the absolute address mode used previously. In the absolute addressing mode, the operand is the address of the data.

The fourth instruction in Table 3 is STA COUNT. STA is like STX in that it stores the contents of a register at an address given by the operand. The difference is that this stores the contents of the accumulator, rather than the X register. STA and STX have different op-codes. COUNT is a symbolic address. When the program is assembled, the address 0031_H will be used for COUNT.

DEC is an abbreviation for DECrement and means, "subtract one from the data at an address given by the operand." Thus, DEC COUNT subtracts one from data called COUNT.

BEQ is an example of another type of instruction, a branch instruction. This kind of instruction can take one of two possible routes through the program, depending on some result.

BEQ is an abbreviation for "Branch if EQual to zero." This instruction checks the result of the last arithmetic the CPU performed. In this case, the last arithmetic was performed by the DEC instruction. If the result of that operation did give zero, the program branches. This means that the next instruction is not the next in sequence. The full instruction is BEQ JUMP, which indicates that if the branch is taken, the next instruction is found at the symbolic address JUMP. So, if the DEC instruction does give zero, the program next executes the instruction at JUMP. This restarts the outer loop.

When DEC does not give zero, the instruction following BEQ is executed. This causes the program to cycle back to the label CYCLE where it again decrements COUNT.

The assembly of this program is relatively straightforward. The program was started at 0020. It should be noted that the operand of the LDA# instruction is 08 and is stored in address 0025. To change the delay, the number could be altered. The operand for the BEQ instruction is FB_H, which was calculated as the number to add to the PC if the branch is taken. While the BEQ instruction is being executed by the CPU, the PC is set to the next instruction in sequence – in this case, $2E_H$. If a branch is taken, the operand 03_H is added to $2E_H$ to give the next instruction.

PC value ⟶ $2E_H$ = 0010 1110

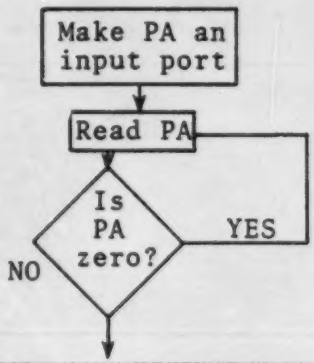
BEQ operand ⟶ 03_H = 0000 0011

0011 0001 = 31_H ⟶ Address of JUMP

EXAMPLE D: BRANCH AND IMMEDIATE INSTRUCTIONS.	
Given:	The following program segment: LDA #00 STA 1701 TEST: LDA PA BEQ TEST
Find:	Its effect and draw a corresponding flow chart.
Solution:	The first two instructions place 00 into 1701. This makes PA an input port. Then, PA is loaded into the accumulator. If the accumulator is not zero, the program jumps back to TEST, where PA is again loaded into the accumulator. As a result, the program will endlessly test PA until all inputs are zero. When at least one PA line is non-zero, the program will go on to the next instruction.

Example D. Continued.

FLOW DIAGRAM:



EXAMPLE E: BRANCH ADDRESSES.

Given: The following machine code:

Address	Op-Code	Operand
0020	F0	07

Find: The address of the next instruction, depending on which branch is taken.

Solution: This instruction is the BEQ branch instruction. If the result of the last operation is zero, the next instruction in sequence will be used. This is at address 0022_H, two bytes beyond the address of this instruction. If the result is not zero, the branch is taken. Then, the operand 07_H is added to the address of the next instruction in sequence (0022_H) to get the following:

$$\begin{array}{r} 22_H \\ 07_H \\ \hline 29_H \end{array}$$

Example E. Continued.

> The result is 0029_H, the address of the next instruction if the branch is taken.

OTHER INSTRUCTIONS

There is a total of 146 instructions the KIM's CPU can execute based on the instruction set of the MOSTECH 6502 chip. A careful study of these instructions is far beyond the scope of this module; however, there are several references included in this module which the interested student can use. In the following paragraphs, two classes of instructions are discussed in general.

The only arithmetic operations discussed to this point involve adding and subtracting one. The CPU can also add or subtract any two single-byte numbers. However, like most 8-bit microcomputers, no instructions are provided for multiplication and division. Therefore, shift left and shift right instructions can be used to result in multiplication and division by 2, as shown in Figure 10. Most 16-bit microprocessors announced after 1977 have built-in multiplication op-codes.

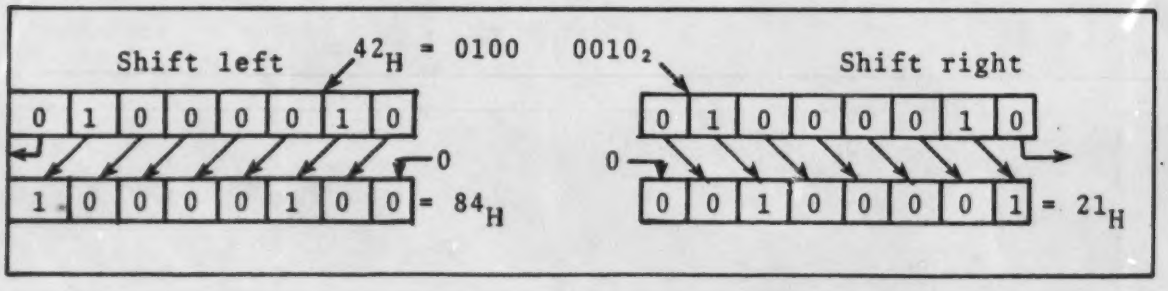


Figure 10. The Effect of Shift Operations.

Addition can also be accomplished by the ADC command, which is an abbreviation for "ADd with Carry." The carry is a bit that is used for multiple-byte additions. To simplify this discussion, this carry bit will be ignored. However, to do this, the carry bit must always be set to zero before using ADC. This is done w.th the CLC command ("CLear Carry").

The ADC command adds the contents of the accumulator to the operand data. The result is left in the accumulator. For instance, if 03_H is in the accumulator and 14_H is in address 0200, then the following instructions,

```
CLC
ADC  00  02
```

result in the sum, 17_H, being placed in the accumulator.

Multiplication and division can be accomplished in a program by combining shift and add operations. For instance, to multiply $1E_H$ by 5, it is shifted left twice to multiply by 4; then the original $1E_H$ is added to get the result, 96_H. This operation is shown in Figure 11. In checking this result, one would convert to the more familiar decimal code. The problem becomes 5 times 30_{10}, which gives 150_{10} — the decimal equivalent of 96_H.

$1E_H$	= 0001	1110	(30_{10})
shift left	0011	1100	(60_{10})
shift left	0111	1000	(120_{10})
add $1E_H$	+ 0001	1110	$(+30_{10})$
result 96_H	= 1001	0110	(150_{10})

Figure 11. Multiplication by 5.

EXAMPLE F: MULTIPLICATION BY ADDS AND SHIFTS.	
Given:	An assembly code that results in multiplication.
Find:	Multiply a number in MULT by 9, using adds and shifts.
Solution:	Nine is 8 + 1, so multiplication by 9 is the same as multiplication by 8 and 1 addition. Multiplication by 8 is done by three left shifts since each multiplies by 2. The following code does it:

```
LDA  MULT     Put MULT in accumulator
ASL  A        Multiply by 2 }
ASL  A        Multiply by 2 } Multiplies by 8.
ASL  A        Multiply by 2 }
CLC           }
ADC  MULT     } Add MULT to result. Answer is in accumulator.
```

A second group of computational op-codes are called logical operations, which include OR, AND and XOR (eXclusive OR) logic operations. Each operation is applied to 2 bits. If these 2 bits are called p and q, then "p OR q" is logical one, if either p or q is logical one. Similarly, "p AND q" is one, if p and q are both one. Finally, "p XOR q" is one, if p or q, but not both, is one.

In 8-bit computers, these logical operations are applied to 8-bit data. In this case, the operations are performed bit-by-bit as illustrated in Figure 12. In this illustration, each bit of the result is found by OR-ing, AND-ing or XOR-ing the corresponding bits of 91_H and $D4_H$.

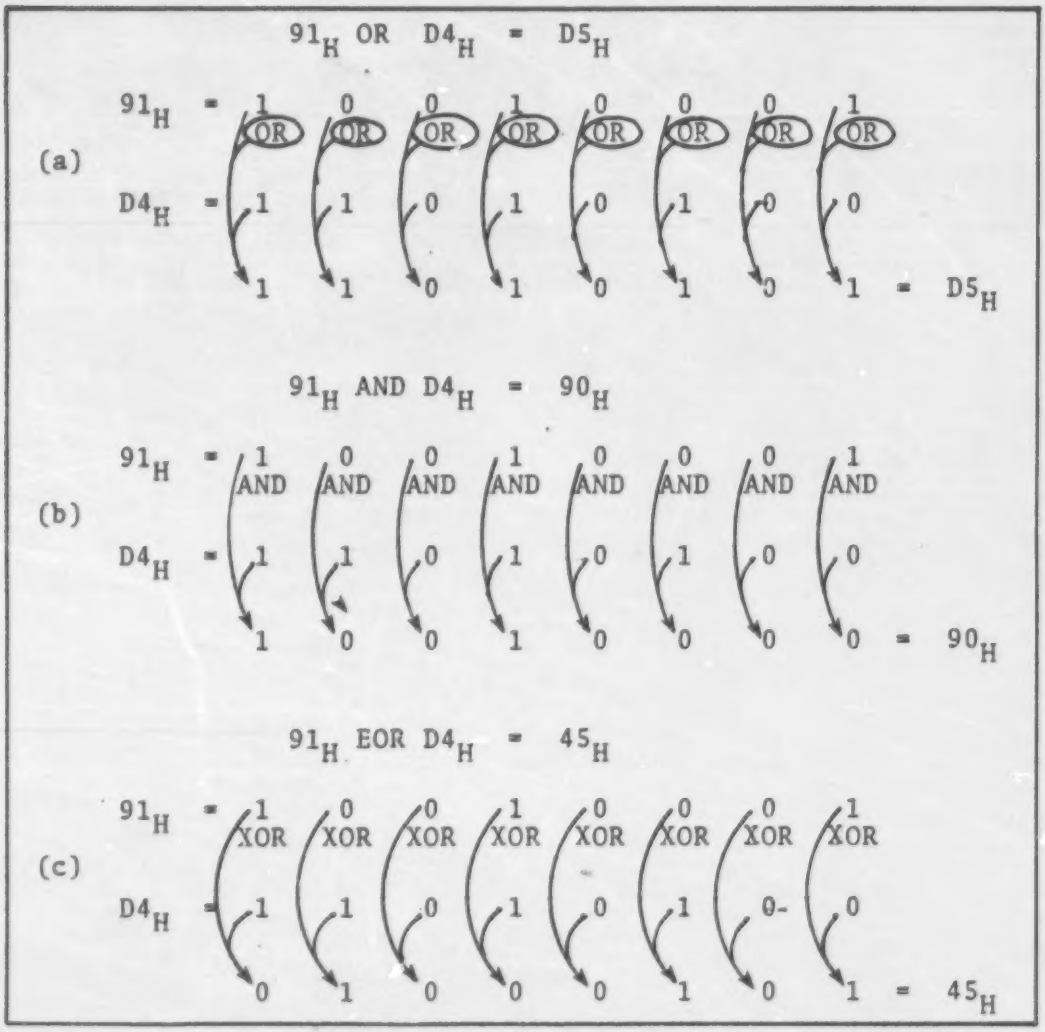

91_H OR $D4_H$ = $D5_H$
(a)
91_H = 1 0 0 1 0 0 0 1
OR OR OR OR OR OR OR OR
$D4_H$ = 1 1 0 1 0 1 0 0
1 1 0 1 0 1 0 1 = $D5_H$
91_H AND $D4_H$ = 90_H
(b)
91_H = 1 0 0 1 0 0 0 1
AND AND AND AND AND AND AND AND
$D4_H$ = 1 1 0 1 0 1 0 0
1 0 0 1 0 0 0 0 = 90_H
91_H EOR $D4_H$ = 45_H
(c)
91_H = 1 0 0 1 0 0 0 1
XOR XOR XOR XOR XOR XOR XOR XOR
$D4_H$ = 1 1 0 1 0 1 0 0
0 1 0 0 0 1 0 1 = 45_H

Figure 12. The Logical Operations, OR, AND and XOR.

Figure 12 gives the following information:

Figure 12a - The logical operation OR. Each bit in the result is 1, if the corresponding bit of either 91_H or $D4_H$ is 1.

Figure 12b - The logical operation AND. Each bit in the result is 1, if the correspond-

ing bit of 91_H and $D4_H$ are <u>both</u> 1.

Figure 12c - The logical operation XOR. Note the left-most bit in Figures 12a and 12b. The result is 1 for either an OR or an AND operation. The XOR operation allows the OR operation, but excludes the AND operation; thus, the left-most bit XOR operation result is 0.

EXAMPLE G: LOGIC OPERATIONS.	
Given:	The numbers $4C_H$ and $1D_H$.
Find:	The result of applying AND, OR and XOR to these two numbers.
Solution:	4C = 0100 1100 1D = <u>0001 1101</u> AND = 0000 1100 = $0C_H$ 4C = 0100 1100 1D = <u>0001 1101</u> OR = 0101 1101 = $5D_H$ 4C = 0100 1100 1D = <u>0001 1101</u> XOR = 0101 0001 = 51_H

1. What digital number produces 13.1 V from a DAC which generates 0.08 V per step?
2. What voltage is produced by a DAC which generates .010 V per step when the binary equivalent of $17F_H$ is applied?
3. Assemble the following code into machine code, starting at 0050_H. Use address 0310_H for SARAH.

```
SAM:  LDA   #FF
      STA   SARAH
      JMP   SAM
```

4. What is the address of the instruction executed after the following?

ADDRESS	OP-CODE	OPERAND
00F0	A9	01
00F2	F0	20

5. Write the assembly code that multiplies the contents of BILL by three.
6. Find the result of the following (in hexadecimal):
   a. 01_H OR 30_H
   b. 78_H AND 87_H
   c. 47_H XOR FA_H
   d. 77_H OR 11_H

LABORATORY MATERIALS

Microcomputer (Commodore KIM-1).
Power supplies: 5 volts at 1 A (TERC PS-005)
+12 volts at 100 mA (TERC PS-012).
Cassette tape recorder (Sanyo ST-45).
Software on cassette tape.
Connections to KIM output ports, power and tape recorder.
Breadboarding system (TERC KIM-100).

LABORATORY PROCEDURES

The first laboratory explores the primary architectural features of the KIM-1 microcomputer. RAM is used to store data and ROM is examined. Then a program is loaded and executed one step at a time. The effect of each instruction is predicted and checked against the KIM's operation.

The second laboratory examines the effect of additional instructions. This level of understanding is important in tracking down program errors. A program with errors is then examined and corrected.

IMPORTANT: Read entire paragraph(s) before carrying out each numbered instruction.

LABORATORY 1: KIM ARCHITECTURE.

1. Make the connections to the KIM-1 shown in Figure 13.
All connections to the KIM are made through the KIMBOARD. ALWAYS CONNECT GROUNDS FIRST. Run a wire from the ground connector on the KIMBOARD (it has the legend "GND") to the minus and ground terminals on the +5 V power supply.

If more than one power supply is used, interconnect all their grounds. Next, make sure the power supplies are OFF. Then connect the 5 V supply to the +5 V terminal on the KIMBOARD. Connect the +12 V supply to the +12 V terminal. Plug the KIM-1 into the KIMBOARD. Finally, connect the interface board to the KIMBOARD, using the 20-conductor pink ribbon cable. Be sure to insert the two white connectors the same side up. Recheck all connections very carefully.

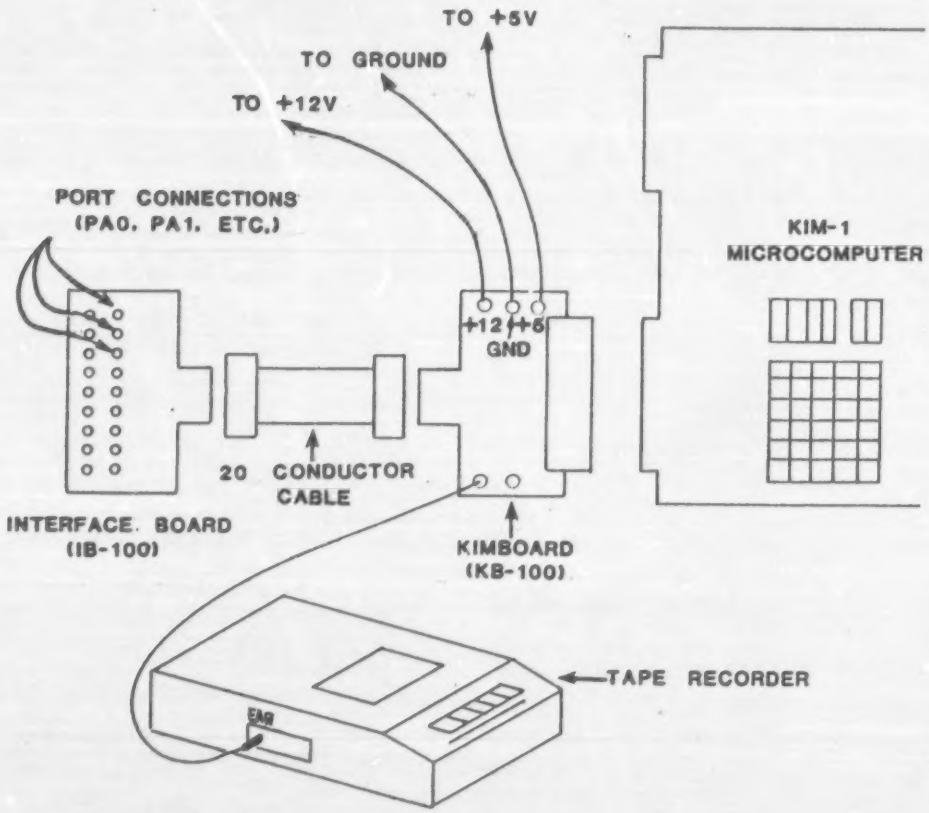


Figure 13. KIM-1 Connections.

2. Turn on and reset the KIM. Apply power to the KIM and press the reset button, marked "RS" on the keyboard. The six-digit display should light. If it does not, quickly remove the power and get help.

3. Read ROM memory. Try writing into address 1800_H. (Refer to the instruction in Module MO-01 for help in using the KIM to do this.) It is not possible because 1800_H addresses read-only-memory. What is observed? Read and record in Data Table 1 (ROM Memory) the data in the 16_{10} addresses, starting at 1800_H. Try to read the data in address 1000_H. Is this a ROM address? Read and record in Data Table 1 (ROM Memory) the data in the 16_{10} addresses, starting at 1000_H. How is this different from the data at 1800_H? Is it possible that neither ROM nor RAM is at these addresses? Explain.

4. Find RAM memory. Verify that addresses 0300_H and 0301_H contain RAM memory by writing 00_H into both and then checking back to see if the data is unchanged. Repeat with FF_H as data. Now see if 0400_H contains RAM by trying to store 00_H and FF_H at that address. What is observed? Record any observations in Data Table 1 (RAM memory). Use this procedure to find the addresses of all available RAM memory. (Hint: Some RAM memory is within the range 1750_H to 1800_H.)

5. To make port A an output port, FF must be stored in address 1701. Then change address 1700 to 00 to assure a 00 count for starting the program in Step 6.

6. Load in the program in Table 1. This program starts at 0010 with the op-code E8. The successive bytes in the program, starting with the first, are as follows:

E8	8E	00	17	4C	10	00
INX	STX	Port A		JMP	0010	

The last 00 should be in address 0016_H. Enter this program and double-check that it is correct by reading

through all 8 bytes.

7. Single-step through the program. The computer can be stopped after it performs each instruction. Do the following:
   a. Put 00_H into address $17FA_H$.
   b. Put $1C_H$ into address $17FB_H$.
   c. Slide the single step switch on the keyboard to the ON position.
   d. Display address 0010_H, the starting address.

   Now each time [GO] is pressed, the computer will execute the instruction indicated by the op-code displayed in the right pair of digits. Press [GO] repeatedly. Only three instructions should be seen – E8, 8E, and 4C alternately. Find out what the program does to the PA lines by doing the following:
   a. Measure the voltages on the 8 PA lines.
   b. Convert these voltages to a hexadecimal number and record the result.
   c. Press GO three times in single step mode.
   d. Repeat Steps (a), (b), and (c) three times.

   Instead of measuring voltages in Steps (a) and (b), directly read the hexadecimal value in address 1700_H. After this is done, press [PC] , before pressing [GO] , to execute the next step. What conclusion is made concerning the net effect of the program?

8. Examine the effect of each instruction. Now examine the effect of each instruction in detail. Before starting, load the X index register with 00. This can be done by placing 00_H at address $00F5_H$. Before executing a step, predict the effect of that step on the X index register and port A. Next, execute the step and then examine the accumulator X index register and port A (addresses $00F5_H$ and 1700_H). To execute the next step, press [PC] to

restore the PC register to the display, and then press GO. Repeat the process of predicting the effect of each step and then checking the result for 10 instructions, starting at address 0010_H. Record all results in Data Table 1 (Single Stepping).

LABORATORY 2: INSTRUCTIONS AND TROUBLESHOOTING.

1. Wire up the KIM and apply power.
2. Load the test program from tape. A cassette tape containing the program illustrated in Table 4 will be provided. Load the program into the KIM.

TABLE 4. A TEST PROGRAM.

Assembly Code			Address	Machine Code
	LDA	#00	0300	A9 00
	STA	COUNT	0302	8D 00 00
START:	LDA	#FE	0305	A9 FE
	STA	VAL	0307	8D 01 00
	STA	1701	030A	8D 01 17
	LDX	L	030D	AE 02 00
TEST	LDA	#01	0310	A9 01
	AND	1700	0312	2D 00 17
	BNE	TEST	0315	F0 F9
LOOP	INC	COUNT	0317	EE 00 00
	LSR	VAL	031A	4E 01 00
	DEX		031D	CA
	BNE	LOOP	031E	DO F7
	JMP	START	0320	4C 05 03

3. Single-step through the program. Before starting the program, load 2_H into address 0002_H. Then single-step through the program, starting at address 0300_H. Before each step, predict the effect of the step; after each step record the contents of addresses 0000, 0001_H and the A and X registers. Before executing the code at address 0312_H, attach a wire from PA to ground. What effect does this have on this step? Disconnect the wire and continue single-stepping for a total of 20 steps. Record all data in Data Table 2 (Single Stepping).

4. Find the errors in Table 5. As another exercise, try to find errors in the assembly of the section of a program listed in Table 5. Perhaps they can be found through close examination; but even if they are, load the program into the KIM and try to single-step through it. How can errors be recognized using single-stepping? Find and correct the errors. Record answers in Data Table 2 (Troubleshooting).

TABLE 5. CODE WITH ERRORS.

Assembly Code	Address	Machine Code
START: LDA #07	0200	AD 07
AND 1700	0202	2D 00 17
BEQ START	0205	F0 FA

DATA TABLES

DATA TABLE 1: KIM ARCHITECTURE.

STEP 3: ROM MEMORY

What happens when writing at 1800_H? _____

_____

Data starting at 1800_H:

ADDRESS	DATA

ADDRESS	DATA

Is 1000 a ROM address? _____ Justify: _____

_____

Describe the data starting at 1000_H: _____

Explain any observations: _____

_____

STEP 4: RAM MEMORY

Does 0400_H contain RAM? _____

RAM addresses in the range 1750_H – 1800_H: _____

_____

Explain the procedure: _____

_____

Data Table 1. Continued.

STEP 7: SINGLE STEPPING
Record all data below:

STEP	ADDRESS	OP-CODE	VALUES IN			DESCRIBE OPERATION
			A	X	PA	
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						

TABLE 2. INSTRUCTIONS FOR TROUBLESHOOTING.

STEP 3: SINGLE STEPPING
Record all data below:

STEP	ADDRESS	OP-CODE	VALUES IN A	VALUES IN X	VALUES IN 0000	VALUES IN 001	DESCRIBE OPERATION
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							

STEP 4: TROUBLESHOOTING

How did single-stepping help find the errors? _____

_____

_____

Data Table 2. Continued.

What is the correct machine code for this assembly program?

ASSEMBLY CODE	ADDRESS	MACHINE CODES
START: LDA #07	0200	
AND 1700		
BNE START		

REFERENCES

Caxton, Foster. Microcomputer Programming: The 6502. Reading, MA: Addison-Wesley Publishing Company, 1978.

KIM-1 User's Manual. Norristown, PA: MOS Technology, 1977.

Larsen, et al. The Bugbooks. E & L Instruments, 1975.

Lin, Wen C. Microprocessors: Fundamentals and Applications. IEEE Press, 1977.

Rao, Guthikonda. Microprocessors and Microcomputer Systems. New York: Van Nostrand Reinhold Company, 1978.

R6500 Microcomputer Programming Manual. Rockwell International, 1978.

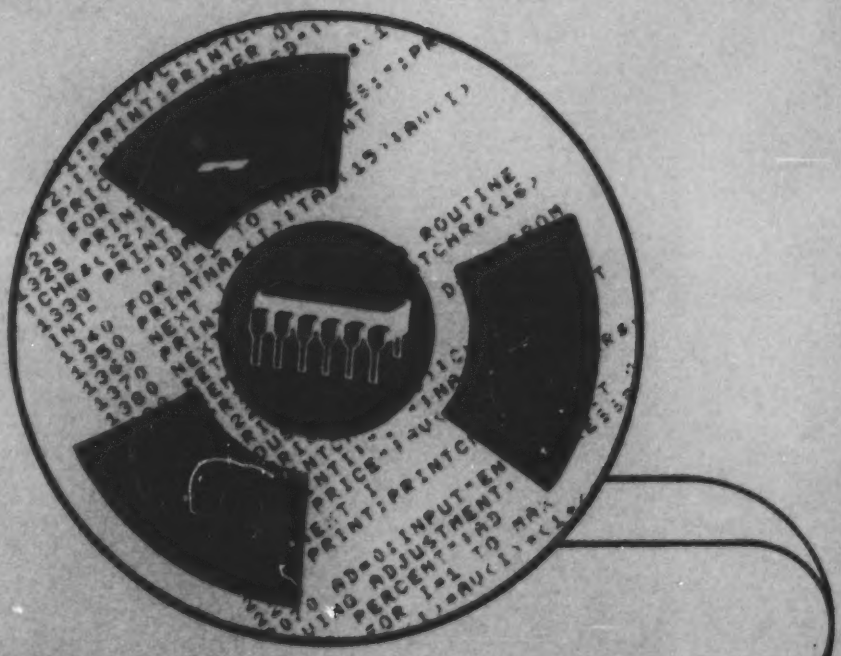
Soucek, Branko. Microprocessors and Microcomputers. New York: John Wiley & Sons, Inc., 1976.

Waite, Mitchel, et al. Microcomputer Primer. Sams, 1976.

Zaks, Rodnay. Programming the 6502. Sybex, 1979.

ENERGY TECHNOLOGY

CONSERVATION AND USE



MICROCOMPUTER OPERATIONS

MODULE MO-03

MICROCOMPUTER APPLICATIONS

CENTER FOR OCCUPATIONAL RESEARCH AND DEVELOPMENT

INTRODUCTION

This module addresses two areas of applications particularly well-suited for small microcomputers. The first type of application involves controlling machinery. Almost anything can be controlled by a microcomputer: stage lights, power generating plants, a bank of elevators, traffic lights, a home-heating system. This last example, a home-heating system, will be explored in some detail in the course of this module since it represents a typical microcomputer-control application in the area of energy-conservation.

The second type of application to be examined involves data logging. In data-logging applications, data are recorded over a period of time for later analysis. For example, a microcomputer-based data-logging system would be useful in predetermining energy needs at a particular building site. By monitoring variables, such as temperature, wind velocity, solar radiation, etc., in advance and over a long period of time, this system can store much of the important information necessary to the design of efficient space-heating for the intended structure. Laboratory work in this module includes using a data logger.

PREREQUISITES

The student should have completed Modules MO-01 and MO-02 of Microcomputer Operations.

OBJECTIVES

Upon completion of this module, the student should be able to:

1. Distinguish between control and computational applications of microcomputers; describe the hardware and software requirements for each type of application.
2. Set up and operate a microcomputer system that could do the following:
   a. Control a home solar heating system.
   b. Log temperature, light and wind data.
3. Define the following terms:
   a. Data-logger.
   b. Transducers.
   c. Feedback.
   d. Analog conditioning circuits.
   e. Controlled system.
   f. Controller.
   g. Instability.
   h. Sensors.
   i. Actuators.
   j. Site evaluation.
   k. Sampling rate.

CONTROL APPLICATIONS

Three typical control systems are used as examples in the material that follows. These systems control a pair of elevators, traffic lights at an intersection and a home solar-heating system.

The elevator controller controls the motion of two elevators in response to buttons pushed by passengers. For each elevator, the controller also opens and closes the door, lights the appropriate indicators that show which floor the elevator is on, and rings a bell just before it arrives at a floor.

A sophisticated traffic light controls the flow of pedestrians and traffic by lighting the appropriate red, green, and amber "walk" and "don't walk" lamps. The controller responds to pedestrians through buttons which are pressed by pedestrians who wish to cross the street; it responds to traffic through buried metal detectors; and it also responds to an electronic clock which tells it when to expect rush-hour traffic.

The solar-heating system controller provides heat when the building it controls gets too cool. It must also store (in a storage bin) any heat generated by collectors and decide whether to use that heat, or heat from an auxiliary heater, to warm the building.

SYSTEM ELEMENTS

The four elements illustrated in Figure 1 are part of any control system. First, there is the controlled system, the objects which are to be controlled; in the examples,

these are the elevators, traffic lights, or the building temperature. The second element consists of sensors, or transducers, which are used to determine if any control adjustments are necessary. In the elevator example, these are a series of switches along the elevator shaft which sense where the elevators are and the buttons that passengers press; in the traffic problem, these are the detectors that are buried in the ground that sense when a car is near, the pedestrian cross-request buttons, and the clock. In the home heating problem, the sensors detect the temperature in the room that is being controlled.

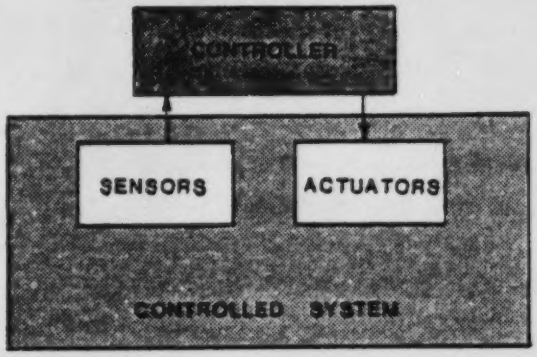


Figure 1. Elements of a Control System.

The third element in a control system is the controller itself, which is usually electronic circuitry that is capable of making control decisions based on the sensor input. In the elevator example, the controller decides whether any eleveator is to go up or down, how far it is to go, where it is to stop, and whether it is to open or close its doors. The controller bases its decision on what it knows about where the elevator is and where it should be according to the requests that have been made of it. In the traffic light example, the controller

controls the turning on and off of traffic lights based on traffic flow and other information fed to it through its sensors. Finally, in the example of the home heating system, the controller determines whether the house is to be warmed and whether there is heat available from solar collectors; and, if so, where it should go, In all of these cases, the controller can be a microcomputer. Today, microcomputers are usually used as controllers, because they lower the cost while adding flexibility and sophistication.

The last element in the control system are the actuators. These are the devices which actually cause changes in the controlled system. In the case of the elevator example, the actuators are the motors which raise and lower the various elevators and open and close the doors. In the traffic light example, the actuators are the traffic lights themselves and their associated circuitry. In the example of the home heating system, the actuators are the fans that circulate the air and the various baffles that can be opened and closed to determine where that air goes.

CONTROL THEORY

The control system in Figure 1 is in the form of a loop, often called a feedback loop. The controlled system affects the sensors, and the controller uses this input to affect the controlled system through the actuators.

Anytime a feedback loop exists, there is a chance for instability where the controller causes large swings in the controlled system. An example of this instability could occur in a room with a heater and an air conditioner. If the controller were improperly designed, a drop in temperature could trigger a blast of heat. If there is a delay in sensing this, the room could get quite hot before the air conditioner

was switched on. This delay could then freeze the room until the controller noticed it and again called for heat.

Thus, a poor controller can be very inefficient if it is unstable. The study of feedback in control systems, with the purpose of predicting and avoiding instability, is the subject of the control theory; however, this important area of study is beyond the scope of this module. (The student should be aware that feedback can result in instability, under certain conditions).

THE SOLAR-HEATED HOUSE

It is instructive to examine the solar-heated system in some detail. Before beginning this, however, it is necessary to understand something about solar heating and some of the control problems involved.

The major difficulty with solar heating is that it is erratic; therefore, it is necessary to store the heat generated by the sun so that it can be used at a later time when there might not be sunlight available. Furthermore, it is expensive to build a solar system that is large enough to meet the energy demands of several consecutive cloudy days. The most efficient approach is to have a system that can store energy for two or three days, and back up that system with conventional heating that can be called upon in those times when there is insufficient solar heat generated. As a result, most solar-heating systems are designed to cover 40 to 80% of the space-heating needs of a building and are backed up with a conventional heating system for the remaining load.

A controller for a solar-heated house has several functions. It must do the following:

1. Feed heat into the storage bin from the collectors whenever possible.
2. Heat the house when necessary.

3. Use the back-up heater only when it must.

Figure 2 illustrates a simplified hot-air solar system. A solar collector on the left warms the air, and the temperature is monitored by an electronic thermometer (indicated by T_C). A fan can direct this warm air to either a heat storage bin or to the house, depending on the setting of four baffles (labeled B_1 through B_4). If B_1 is open and B_2 is closed, the fan draws air from the collector; when B_4 is closed and B_3 is opened, this hot air is directed to the storage bin. The storage bin temperature is monitored by an electronic thermometer (T_B). Heat is collected only when the collector air temperature is higher than the storage bin temperature.

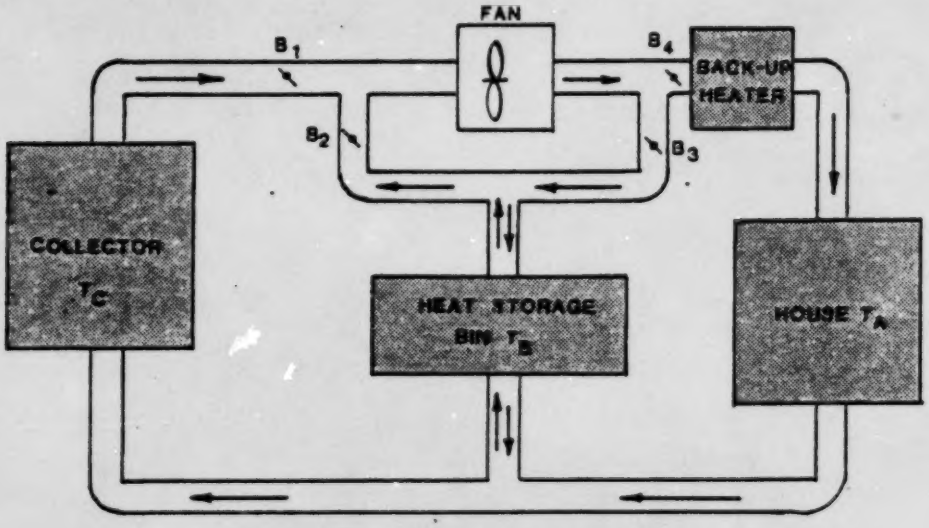


Figure 2. Simplified Solar Control System.

House temperature is monitored by the thermometer (T_A). To heat the house, B_4 is opened and B_3 is closed. The fan draws air from either the collector or storage bin (whichever is warmer) by opening either B_1 or B_2. If the air is not sufficiently warm, the back-up heater is turned on to

warm the air temperature enough to heat the house.

The four elements of this control system are as follows:

1. Controlled: House and bin temperature.
2. Sensors: Three electronic thermometers; T_A, T_B, T_C.
3. Controller: A microcomputer.
4. Actuators: Fan, back-up heater, and four baffles; B_1, B_2, B_3, B_4.

Figure 3 illustrates the information used by the controller (analog temperature from the three sensors) and the information generated by the controller (on/off information for the six actuators).

In the laboratory section, the KIM microcomputer is programmed to perform this control operation. Instead of using real equipment, the heating system is simulated by a second program. The computer simulates reasonable values for the temperatures, T_A, T_B, and T_C. The control program then turns on and off the actuators which are simulated by lights.

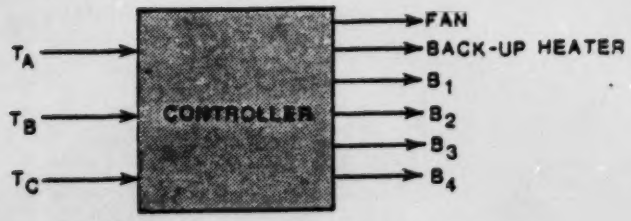


Figure 3. Logic Flow of Solar House Controller.

To show what the controller does, the controller can be placed on manual so the student must control the system.

The program can then score how well a student does as a controller. This can be made into a game in which the student tries to beat the computer; that is, do a better controlling job than the computer.

The rules of the game are as follows:

1. If the room temperature is below 65°F, then heat is required until the room reaches 70°F.
2. Room heat is drawn from the collector or the storage bin, whichever is warmer.
3. The back-up heater is required if the air entering the house would be below 80°F without the heater.
4. The collector should warm the storage bin if the collector is warmer than the bin and the house is not being heated.
5. The fan should be off if neither the house nor the heat storage bin is being warmed.

The computer simulates a cold and partly cloudy day, and displays the three temperatures, T_A, T_B, and T_C. On the basis of these three changing temperatures, the student must decide how to adjust the four baffles, the fan, and the back-up heater. The status (on or off) of each is indicated by a light and can be changed by pressing a button. Anytime incorrect choices are made, an error light comes on.

The purpose of these programs is to illustrate that a small microcomputer can be used for control. This example is slightly artificial since the computer does both jobs; it performs the control and simulates the controlled system. In reality, the simulation would not be needed - which is a factor that would somewhat reduce the computation required as compared to the example. On the other hand, real data would be digitized with the computer (as is done in the next data-logging example), which requires more hardware and programming than was required in the solar-house controller

example. Simple control programs are similar in size to the one used in the lab and fit easily into a small computer such as the KIM.

DATA LOGGING

Data logging is used whenever it is necessary to record data for later analysis. The data is usually accumulated slowly over a period of time.

A data-logging device would be useful in the evaluation of a particular solar panel, for example. To evaluate the solar panel, the device would measure the temperature, air flow, and sunlight levels around the collector over a period of many days; then a microcomputer would automatically record this data and store them in its memory. The data could then be read out and evaluated at a later time.

Another example of data logging involves predetermining and evaluating energy needs at a particular building site. To design efficient space heating of a structure, it is important for designers to obtain certain information, such as average temperature, wind velocity, solar radiation, and radiative cooling, from the intended building site itself. This data is best obtained by actually monitoring these variables at the intended site over a long period of time. A microcomputer-based data-logging system can be used for this.

As an example of this capability, a data logger - using the KIM microcomputer - will be used in the laboratory to measure temperature. With slight modifications this logger could also be adapted to measure the other variables required for site evaluation.

LOGGER COMPONENTS

Most microcomputer data-logging systems contain the components shown in Figure 4. A sensor, or transducer, converts a physical input, such as light or temperature, into an electrical signal. This signal is usually too small to be used directly, but analog conditioning circuits are used to convert the signal into a form that can be used. A D-to-A (D/A) convertor generates a 1-byte digital equivalent of the analog signal from the sensor. The microcomputer regularly samples this digitized signal and stores the corresponding bytes in Random Access Memory (RAM). The rate at which signals are recorded is called the sampling rate. The recorded samples can be recovered later, using some kind of display device.

Figure 4. Data-Logging Components.

In the laboratory, the sensor will be in the form of an electronic thermometer. The signal from this is amplified and digitized. Five hundred and twelve samples are taken at various rates. The results are displayed as a graph of temperature against time.

The sampling rate determines the total time recorded, since 256 samples will always be taken. If five samples are taken every second, only 51.2 seconds can be recorded. With samples taken every 20 minutes, more than two days of

data can be recorded.

The sampling rate should be adjusted to the time scale of the fastest changes to be recorded. Since air temperature rarely changes significantly over a 20-minute period, this sampling rate is sufficient. Sunlight can change over a few seconds, as clouds shift; therefore, it would require a different sampling rate.

Because the laboratory cannot be extended two days, some quick-changing temperatures will be recorded at high sampling rates. The same apparatus could be used at low rates for site evaluation.

SITE EVALUATION

A full site evaluation requiring the measurement of several variables over a period of several months would require much more memory. The conventional approach to this is to store the data on magnetic tape. Even a small cassette tape can store hundreds of kilobytes. With a simple addition, the KIM can control a cassette tape recorder and store the data on tape; this frees RAM memory for temporary data storage, which is then permanently recorded on tape. Thus, the 1-kilobyte memory limitation does not limit the amount of data that could be obtained from a site evaluation.

The above example illustrates the utility of using microcomputer-based data-logging. The program can be stored in less than 500 bytes of memory and the sampled data fills 512 bytes.

LARGER SYSTEMS

The student may wonder at this point why larger microcomputers are used at all if a small microcomputer like the KIM can be

used in so many diverse applications. This is because larger systems have three important features the KIM lacks:

1. Computational power.
   The applications discussed require little or no calculations. This is fortunate because the KIM, like most microprocessors, can only add and subtract single bytes. Multiplication functions and floating point precision are obtained only through programs that may occupy many kilobytes.
2. Flexibility.
   The KIM programs used are fixed-purpose and the result of much programming effort at the assembly level; as a result, they are difficult to change. Furthermore, the KIM can be of little help in the programming process. Larger machines can assemble code and be programmed in interactive, high-level languages, like BASIC, that make program development and modification much easier.
3. Mass storage.
   The cassette tapes KIM uses are not a convenient way to store many programs because they are slow and not under direct computer control. Floppy disks are used with larger systems to get around these limitations and provide millions of bytes of storage that can be accessed in a fraction of a second.

EXERCISES

1. Which of the following tasks could be accomplished by a small microcomputer like the KIM? Justify the answers given.
   a. A computer that would operate the traffic lights at an intersection.
   b. A computer that tells a company when to deliver heating oil to customers.
   c. A computer that could determine tax and withholding on paychecks for a small company.
   d. A computer that would allow an operator to enter address labels for a magazine and then type them out in zip code order.
2. What would be needed, in addition to the CPU, for each application listed in Exercise 1.

LABORATORY MATERIALS

Laboratory 1 and 2

KIM-1 microcomputer.
Oscilloscope with external triggering.
Cassette tape recorder.
Digital voltmeter (optional).
Cassette tape with cooling curve program.

PARTS

ICs: 741 opamp.
3140 opamp.
311 comparator.
1408 D/A converter.

Fixed Resistors: 3 1kΩ.
1 33kΩ.
1 6.8kΩ.
7 470Ω.
1 1.5kΩ.

Variable Resistors: 1 10kΩ.
1 1kΩ.

Capacitors: 1 .005μF.
1 30pF.

7 LEDs
1 diode with Teflon insulation on leads.
1 16-pin DIP switch.

LABORATORY 1. CONTROL.

1. Construct the display.

As discussed briefly in the text, the first laboratory exercise will be to build a solar-home collector. A special display, consisting of seven lights (lamps), called LEDs, is needed to indicate the state of the actuators. This should be constructed and connected to the KIM, as shown in Figure 5.

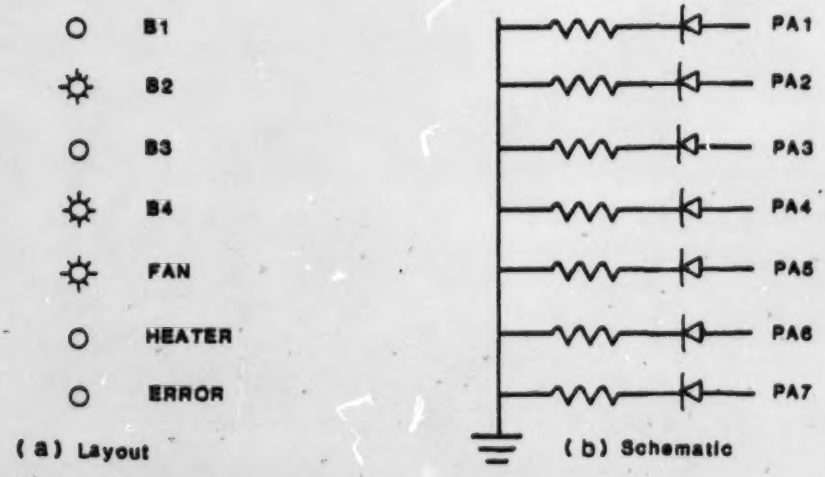


Figure 5. Actuator Status Display.

When the lamps are on, they have the following significance:

B_1 through B_4 - When each of these lamps is on, the corresponding baffle (as labeled in Figure 2) is open.

FAN - When the lamp is on, the fan is on.

HEATER - When the lamp is on, the heater is on.

ERROR - When incorrect choices of actuators are made, the bottom lamp comes on.

The solar house controller simulation in Figure 5 indicates that B_1 and B_3 are closed, B_2 and B_4 are open, the fan is on, the heater is off and that there are no errors.

2. Load the programs from cassette tape.

Apply +12 V to the KIM and attach the tape player cord to the KIM's tape input.

The ID number for the solar-home programs can be found on the tape. Enter this number in address $17FA_H$, then do the following:

Clear $00F1_H$.
Start the tape load program at 1873_H.
Play the tape at full volume and full treble.

When the KIM indicates address 0000, the tape has read properly. If address FFFF comes up, or no address appears after 5 minutes, try again.

3. The manual controller game.

As soon as the program at 0000_H is started (by pressing GO), the computer starts simulating a faster version of a day in the life of a solar home. The student must operate the computer with a manual controller and must make the correct choice of actuator settings in response to changing sensor inputs. The sensors sense the three temperatures, T_A, T_B, and T_Ck as labeled in Figure 2 of the text.

To see what these sensors detect, press the A, B, or C buttons on the KIM and the seven segment display on the KIM will indicate one of the temperatures. (Figure 6 illustrates a temperature on display.) Only one temperature can be displayed at a time, but any of the three are available at any time by pressing the corresponding key - A, B, or C for T_A, T_B, or T_C, respectively.

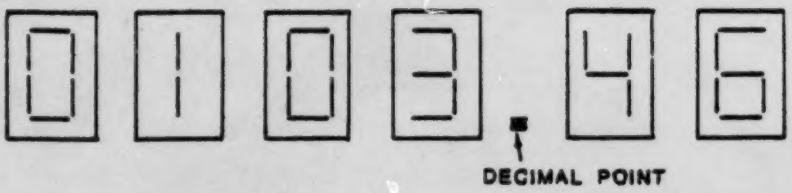


Figure 6. Temperature on Display.

The state of the actuators is indicated on the six LED lights connected in Step 1. To change any of these, press buttons 1 through 6. Each press changes the corresponding actuator from on to off or from off to on.

If the game is played according to the rules in the text, no errors will occur. If mistakes are made, the ERROR light will come on.

4. Run the manual game.

Run the program, starting at 0000 once for practice. Then run it again, trying to control the system. Each run takes about 5 minutes - an increase in speed of about 100 times. When the program finishes a day, the score is displayed. Record this score in Data Table 1 (Controller).

The score is divided into three parts, as illustrated in Figure 7. The left pair of digits indicates the total time that the error lamp was on. The center pair indicates the total time that the house temperature was out of range (65° to 70°F). The right-hand pair indicates the total time fuel was used. (The lower time this shows, the less fuel was consumed.)

Each count in the score corresponds to three seconds of running time, or 15 minutes of simulated time.

Figure 7. Score Divisions.

The rules of the game are as follows:

1. If the room temperature is below 65°F, then heat is required until the room reaches 70°F.
2. Room heat is drawn from the collector or the storage bin, whichever is warmer.
3. The back-up heater is required if the air entering the house would be below 80°F without the heater.
4. The collector should warm the storage bin if the collector is warmer than the bin and the house is not being heated.
5. The fan should be off if neither the house nor the building is being warmed.

5. Run the automatic game.

By altering the contents of address 0003, the computer can be told to control the system. Place 01 into 0003, and restart the program at 0000. Now the same system is simulated, but the computer controls it. When this is done, it displays its score. Record this in Data Table 1 (Controller) and compare it to the other score (from game with manual control).

In most cases, manual control results in less comfortable control (as indicated by the out-of-range time) and more fuel use. Is this the case?

LABORATORY 2. DATA LOGGING.

1. Construct the interface.

The sensor, analog conditioning, and A/D circuits needed for temperature data logging are shown in Figure 8. Consult the instructor for the preferred way of assembling these circuits and attaching them to the KIM.

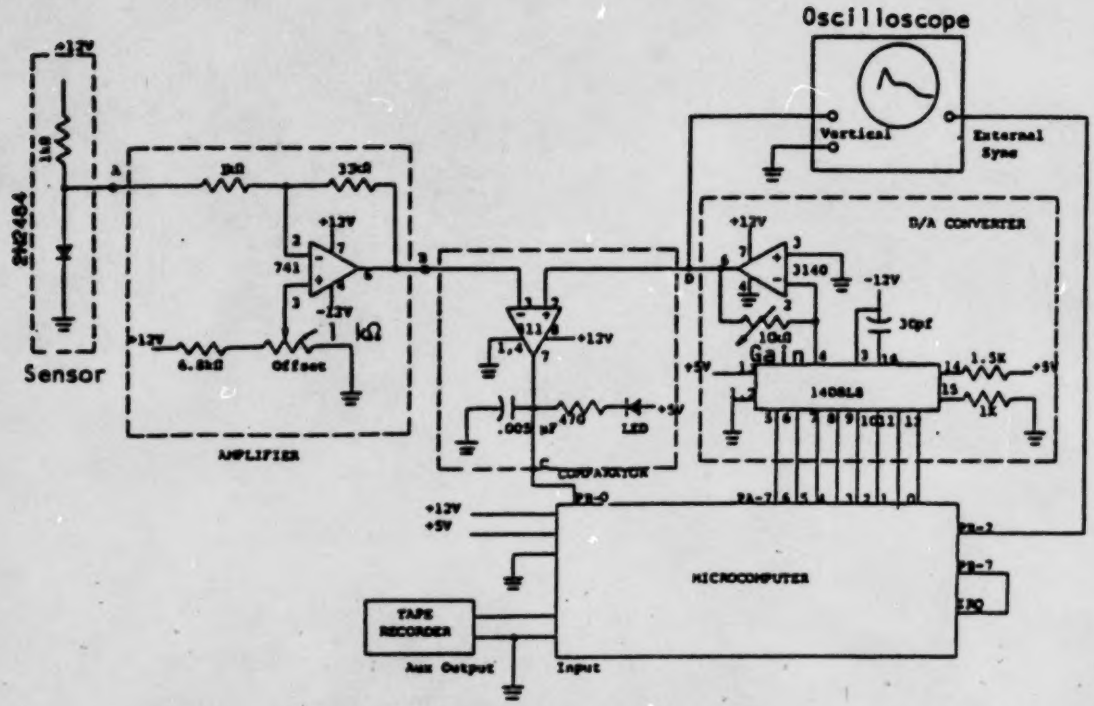


Figure 8. Complete Schematic.

2. Load the program.

The data-logger program is stored on tape. Read it into the KIM and start execution at 0000. The display should immediately display a temperature which can be read by assuming a decimal point after the third digit, as shown in Figure 9. Either Fahrenheit or Celsius units can be used by pressing F or C, respectively; this is displayed in the last digit. Figure 9 shows a temperature of 81.9°F.

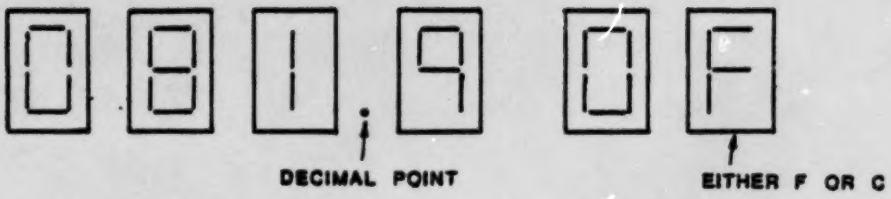


Figure 9. Temperature Display.

3. Log in data.

The microcomputer responds to the following keys:

Press F to start logging data.

Press 1 to see the data displayed on the oscilloscope as it is being logged.

Press 0 to see the results previously logged.

Start logging data, then briefly immerse the sensor in water. Then let the water evaporate and then warm the sensor with body heat (use fingers). Describe the resulting graph. Repeat the process with less water and compare the two graphs. How and why do they differ? Record the answers in Data Table 2 (Data Logging).

DATA TABLE 1: CONTROLLER.

Step 4. Your score:

Error time _____

Out-of-range time _____

Time fuel used _____

Step 5. Computer's score:

Error time _____

Out-of-range time _____

Time fuel used _____

DATA TABLE 2: DATA LOGGING.

Step 4. Describe the graph: _____

_____

_____

_____

Describe differences between two runs:

_____

_____

_____

REFERENCES

Bibbero, Robert J. Microprocessors in Instruments and Control. New York: John Wiley and Sons, Inc., 1977.

Burton and Dexter. "Analog Devices," Microprocessor Systems Handbook, Norwood, MA: 1977.

Foster, Caxton. Microcomputer Programming: The 6502. Reading, MA: Addison-Wesley Publishing Company.

KIM-1 User's Manual. Norristown, PA: MOS Technology, 1977.

McGlynn, Daniel R. Microprocessors: Technology, Architecture, and Applications. New York: John Wiley and Sons, Inc., 1976.

Rao, Guthikonda. Microprocessors and Microcomputer Systems. New York: Van Nostrand Reinhold Company, 1978.

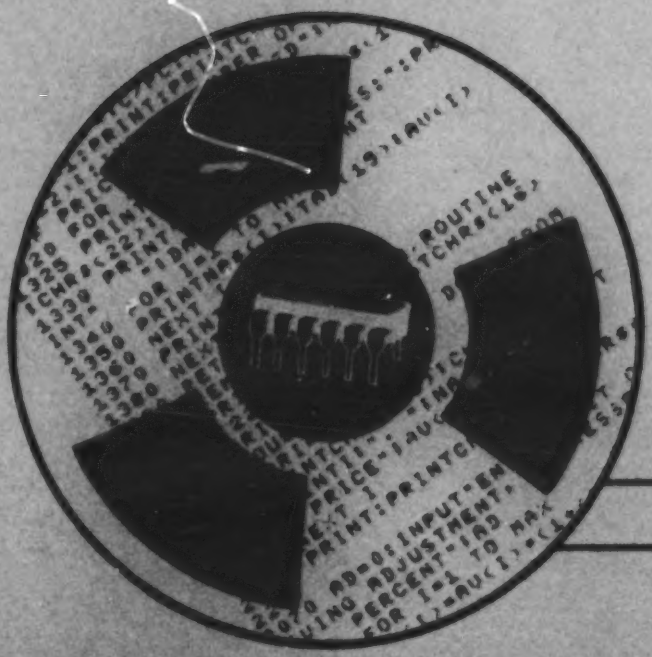
Sippel, Charles J. Microcomputer Handbook. New York: Petrocelli/Charter, 1977.

Sovcek, Branko. Microprocessors and Microcomputers. New York: John Wiley and Sons, Inc., 1976.

Tinker, Robert F. Microcomputers. Cambridge, MA: TERC, 1978.

ENERGY TECHNOLOGY

CONSERVATION AND USE



MICROCOMPUTER OPERATIONS

MODULE MO-04

DISK-BASED OPERATING SYSTEMS

CENTER FOR OCCUPATIONAL RESEARCH AND DEVELOPMENT

INTRODUCTION

A disk-based microcomputer system is simply a system that utilizes a disk for storage. A "disk" is a medium for storage that lends itself to rapid retrieval of large amounts of data. The addition of a disk to a microcomputer system greatly increases its flexibility and power. Much of this increased versatility resides in a series of programs known as the "operating system." This module examines a typical and widely-used operating system, known as "CP/M," which will be used to create, edit, store and execute programs. Computers that the energy technician will encounter in the field will have an operating system that will be able to perform similar functions.

This module is an introduction to the programs in the operating system but does not entail a complete description of all functions.

PREREQUISITES

The student should have completed Modules MO-01 through MO-03 of Microcomputer Operations.

OBJECTIVES

Upon completion of this module, the student should be able to:

1. Use the operating system of a disk-based microcomputer to create, move, edit, rename, display, print, and erase files.

2. Enter, compile, correct, execute, and save BASIC programs that are both interpreted and compiled.
3. Create a personal disk containing a full set of utilities for later use.
4. Define the following terms:
   a. Disk.
   b. Mini floppy disk.
   c. Floppy disk.
   d. Hard disk.
   e. Operating system.
   f. Prompt.
   g. System prompt.
   h. Crash.
   i. Booting.
   j. Cold start.
   k. Warm start.
   l. File.
   m. File name.
   n. Filetype.
   o. COM file.
   p. Editor.
   q. Editor prompt.
   r. Logged in.
   s. Character pointer.
   t. Compiler.
   u. Compile.
   v. Interpret.
5. Describe the function of the following programs:
   a. ED
   b. DIR
   c. TYPE
   d. PIP
   e. SYSGEN

f. REN
g. ERA
h. ASM
i. LOAD
j. BASIC-E
k. BASIC/5

DISK SYSTEMS

Disks add a new dimension to microcomputer systems. The two attributes that make disks so powerful are their large storage capability and relatively fast access. The smallest disks are called mini floppy disks and can store 80,000 bytes of data. The disk system in this module uses so-called full size floppy disks, each of which can store approximately 650,000 bytes of data. Larger, so-called hard disks can store 20, 40, or more million bytes of data. Any of these disks have the ability to write or read into memory within a few hundred milliseconds from any place on the disk.

A disk can be thought of as a record-player-shaped tape recorder. The disk itself is a flat, circular substrate that has on its surface a magnetic material quite similar to that used in tape recorders.

Floppy disks are very fragile, are made from flexible plastic, and "float" on a cushion of air inside a protective paper container. On the other hand, hard disks are made of rigid metal; they are constructed with very tight tolerances so that more information can be placed on their surfaces than the same area of floppy disks.

FILE MANIPULATION

An operating system permits a user to operate or use a computer. This means that the operating system should make it easy to do the following things:

- Write, debug, print, and retrieve locally-generated

programs.

- Load and run programs.
- Load and use one or more high level languages and store programs written in those languages.
- Enter, store, and retrieve data.

By this definition, the ROM monitor in the KIM qualifies as an operating system. However, a much larger operating system can be supported when a disk is part of that system; this makes it correspondingly easier to use the computer.

The operating system is the first program encountered when a computer is turned on, and the operating system must be told what program to retrieve and execute. It asks for this information by displaying the following:

A>

This is computer shorthand for "which program on disk A do you want to execute?" (Disk A is usually, but not always, the right-hand one.) The A> is called a prompt because it tells the user that some response is required before anything further can be done. The computer then halts until the user responds with the name of a program and strikes the carriage return key (RETURN).

When the computer is turned on, the operating system is automatically loaded into the computer from the disk. This is called booting the system from a cold start. The term "booting" is an abbreviation of "picking itself up by its boot straps."

The operating system is also loaded in any time the computer is reset by someone pressing the reset button. This is called a warm start. There are times when some program is running out of control or has inadvertently altered the operating system program in RAM. This is called a system crash and usually can be corrected by pressing RESET.

FILES

A new addressing scheme is required with disks because of their enormous data storage capacity. The random access approach used in the main memory of a computer is neither appropriate nor necessary on a disk. It is inappropriate because the address would have to be long and the access time would be lengthy. It is unnecessary because information can be stored on a disk in large blocks which can be loaded into RAM all at once. This alleviates the problem of accessing each single byte directly.

For these reasons, information is stored on a disk in blocks called files, which can contain varying amounts - from zero bytes to the entire disk (650 kilobytes). Every file has a name that is used instead of an address to gain access to the information within the file. The name of a file must have the following form:

filename.filetype

The filename can have up to 8 characters with no spaces. The following are examples of valid file names: A, MYPROG, and M174$. The name must be followed by a period and a 3-character type. The "type" is useful for telling the operator what type of file it is and for distinguishing related files that have the same name but a different form. With one exception, the operating system normally does not care what the filetype is. The one exception is COM files of the form:

filename.COM

COM files will be discussed later in the section on program execution. The following are valid filenames: MYPROG.AAA,

ED.COM, A.BSC and A3.$$$

CREATING AND STORING FILES

An example is useful at this point to illustrate what a file is and how it is created. Suppose the operator wanted a file called EX.ASC that contained the following phrase:

"This is an example of a file"

The operating system contains a program called an editor, which is used to create or modify files. The editor is started by the operator typing its name (ED) and the name of the file to be edited (EX.ASC is this example). The complete instruction is as follows:

ED EX.ASC <return>

Here, <return> means "strike the RETURN key." (Ret and CR are used interchangeably to represent this.) Return causes the operating system to read the editor into memory from the disk and to tell the editor that the file EX.ASC will be edited. Then the editor searches the disk for a program of that name. Finding none, the disk creates one, adds this name to the directory of names of files, and waits for further instructions. The editor informs the user that it needs input by displaying its prompt, *.

At this point, the file EX.ASC exists in name only; there is nothing in it. In other words, it is a file containing zero bytes. To add to this file, the command "I <return>" (for insert) is typed. After this, anything typed will be added to the file. In particular, the ASCII code of each key pressed

will be added to the EX.ASC file.

Now the following message can be typed: "This is an example of a file". After the last e, ED must be stopped from inserting anything more. This is done by typing the special character CTL-Z, which is an abbreviation for "ConTroL-Z." CTL-Z is generated from the keyboard by pressing both the CONTROL and Z keys together. To emphasize that CTL-Z is one character, the symbol <CTL-Z> will be used hereafter.

The computer now responds with *; this indicates that the editor is again ready for further instructions. If the message is assumed to be correct, it should be saved and the editing session should be ended. This is done by the operator typing e (for end).

Figure 1 shows all the instructions required to file the following message: "This is an example of a file."

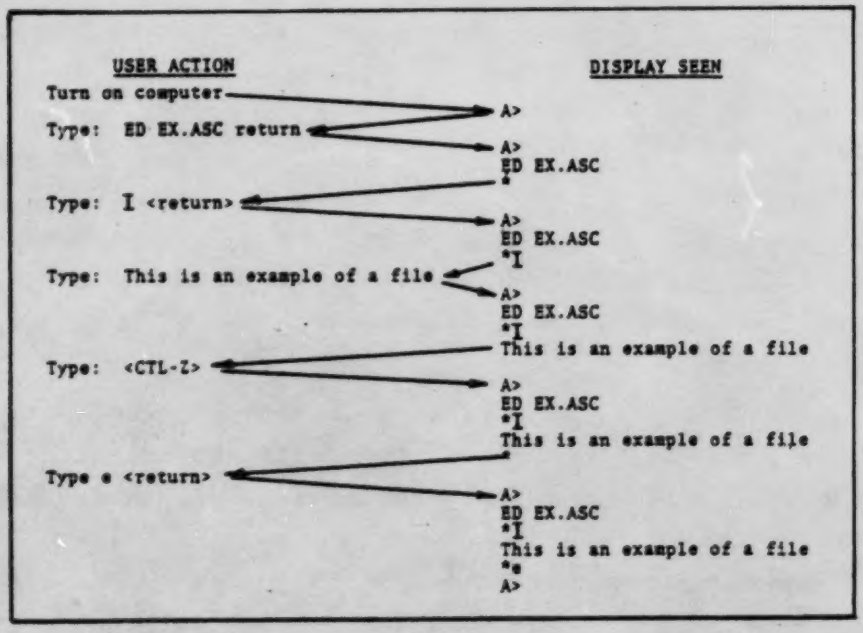


Figure 1. Steps Required to Create a File Called EX.ASC

The editor can be used to modify existing files. This capacity will be examined in a later section of the module.

MISTAKES

When an error is made in entering information, the operator has two options: (1) the current line can be aborted, or (2) the last one or more characters can be deleted. The current line is aborted by typing <CTL-U> (the control and U keys together) and the last character is removed by pressing the DELETE key. The system responds by repeating the deleted letter. (Any number of letters can be deleted this way.)

For an example of the use of the delete option, suppose "THIR" was typed instead of "THIS." After pressing DELETE and S, the operator enters the correct message and the display shows "THIRRS."

FILE CONTROL

At this point it is only assumed that the file EX.ASC exists and contains the intended message. However, the operating system should have the ability to permit the user to verify both the name and contents of any file.

File names can be examined with the program "DIR" (for DIRectory). The file contents can be displayed by using the program "TYPE."

To use DIR, type DIR in response to the system prompt A>; this is the usual way to run a program with the operating system - simply type its name. DIR responds by displaying the names of all files on the disk.

In the present example, the idea is to verify whether a file is on the disk. This can be done after a system prompt by

the operator typing DIR and the file name, as follows:

DIR EX.ASC <return>

The full name is required. DIR will report whether that file is present on the disk. One can also request any file called EX by substituting a star for the type as follows:

DIR EX.* <return>

All ASC filetypes can be requested with the following:

DIR *.ASC <return>

To display the contents of a file, type TYPE; then type the full name of the file. For example, to see the contents of EX.ASC, after a system prompt, type the following:

TYPE EX.ASC <return>

The computer will respond with the message previously entered into this file.

Sometimes the contents of a file are too extensive to fit on the screen; or the contents are displayed so quickly that they are scarcely seen before they are erased again. To stop the display, type <CTL-S> (the control and S keys together). This usually freezes whatever the computer is doing. To continue (or unfreeze the computer) any key can be struck by the operator.

Sometimes the operator must stop a program and reboot the operating system. This could happen when TYPE is displaying a long file and there is no need to wait until it is finished. The command <CTL-C> will usually stop any program and return

to the system.

COPYING FILES

The operating system can copy files from one disk to another. This will be done in the laboratory to create a personal disk from a master disk.

The two disks have the names "A" and "B". If more disks are in the system, they have the names "C," "D," "E," etc. Only one disk can be logged in at a time. The significance of the logged-in disk is that only one normally will be searched for matching file names. For instance, if disk B contains the file X.ASM, then, while the A disk is logged in, a search for that file with the command DIR X.ASM will report finding no file with that name.

To change the disk that is logged in, type the new disk name, followed by a colon. To change to disk B, type as follows:

```
B:
```

To this, the system responds with B>. The currently logged-in disk is always shown as part of the system prompt.

Files can be copied from one disk to another by the operator using the "PIP" program. To copy a file EX.ASM from disk A to disk B, the command is as follows:

```
PIP B: = A:EX.ASM
```

This can be entered in response to a system prompt. All ASM filetypes can be copied with the following:

```
PIP B: = A:*.ASM
```

The equal sign above can be thought of as a left arrow. PIP moves whatever is on the right of the symbol to the left.

The operating system can copy almost anything between disks by using PIP. The only exception is the operating system itself. A special program called "SYSGEN" is used for this. When the program name SYSGEN is typed, the program asks which disk the system resides on – the source. It copies this information into memory and asks onto which disk this is to be read. A blank disk can be inserted to replace the source; or it can be put on the other disk. SYSGEN then copies the system onto the other destination disk. A carriage return gets the system back.

RENAME AND ERASE

It is sometimes necessary to change the name of a file; this done with the program "REN" (for rename). To change the name of file EX.ASC to MES.ABC, the REN command can be used in the following form:

REN MES.ABC = EX.ASC <return>

The new name comes first and is followed by the old name. (Again, think of the equal sign as a left arrow and it should help to remember.)

When a file is no longer needed, it can be erased with the ERA command. To erase the example program, type the following:

ERA EX.ASC <return>

ED, REN, ERA, DIR, PIP, SYSGEN, and TYPE are all commands built into the operating system. Each is a program that can be executed by its name being typed after the system prompts the user. Together they provide full control of files; they can be used to create, store, retrieve, modify, copy, and remove files. The ED program has many additional features that

make it easy to correct and modify files. Some of these features are described in the next section.

EXAMPLE A: FILE CONTROL.	
Given:	The CP/M editor.
Find:	The steps needed to create a file called MES.MES, type it from the disk, change its name to MES.AGE, then erase it. The file should contain the message "The quick brown fox jumped over the lazy dog."
Solution:	ED MES.MES <ret> I <ret> The quick brown fox jumped over the lazy dog. <ret> e <ret> <CTL-Z> TYPE MES.MES <ret> REN MES.AGE = MES.MES <ret> ERA MES.AGE

THE EDITOR

Files can contain any information and might also contain text, as did the example above. A file may also contain data, machine language programs, or any other information used by the computer.

The usual way to develop machine code with a computer is to enter the program in assembly code, then use an assembler to create machine code from the assembly code. In a disk-based system, the assembly code is contained in a file, as are the

assembly program and the resulting maching language. The files that contain different versions of the same program usually are given the same filename with different types. In the example below, a program called DIAG will be written. The assembly version will be called "DIAG.ASM"; while the assembly machine code version will be called "DIAG.COM".

To illustrate the use of the editor, the INTEL 8080 assembly program in Table 1 will be entered into DIAG.ASM. This short program draws diagonal on the TV, starting from the section of the screen.

This program can be entered by using the editor ED in the insert mode. The only new feature is the column format which can be achieved with tabs. CP/M recognizes <CTL-I> as a tab that advances it to the beginning of the next column. A zero and the letter "O" are sometimes difficult to distinguish. Note that in Table 1, the usual convention of slashing zeros is used. The codes in the assembly program are written for 8080 type microprocessors and are different from the codes used in previous modules. At this point, it is not important to understand how the code draws the diagonal. This example is used to illustrate how to create, assemble, and run machine-code programs.

TABLE 1. AN 8080 ASSEMBLY PROGRAM - DIAG.

```
        ORG   1ØØH
START   MVI   A,ØC6H;  Set byte, write direction.
        OUT   19H
        MVI   A,Ø    ;  Set cursor at zero.
        OUT   C
        OUT   1CH
        OUT   1DH
        OUT   1EH
        MVI   B,64   ;  Set counter.
        MVI   A,33H  ;  Set byte command.
LOOP    OUT   1AH    ;  Draw a dashed line.
        DCR   B      ;  Decrease counter.
        JNZ   LOOP   ;  Repeat if not zero.
        JMP   Ø      ;  Return to system.
```

CORRECTING A FILE

If Table 1 had been entered, but contained errors, ED could also be used to correct the errors.

To correct at some location within a file, the operator must be able to specify locations within the file. This is done with an invisible pointer called the character pointer, or CP. Once the character pointer is properly positioned, it can be used as a reference point for removing and inserting characters or whole lines. Thus, the first step in an editing session is to move the CP to the desired place.

The editing of DIAG.ASM is started by typing the following in response to the system prompt A>:

ED DIAG.ASM <ret>

The editor responds with its prompt *. At this point, the file has not been read into memory. Before the file can be changed, it must be read into memory from the disk, which is done with the editor command.

#A <ret>

There is an error in the fifth line in Table 1; the line should read as follows:

OUT 1BH

This means that the C must be removed and 1BH inserted. To do this, the CP must be located just before the C in line 5. Its current location is at the end of the file. The command,

B <ret>

moves the CP to the beginning of the file. Then the command

5L <ret>

moves the CP down five lines.

To check this, the line can be typed with the following command:

T <ret>

The editor will respond with the following line:

<tab> OUT C

At this point, the line can be removed with the "kill"

command:

K <ret>

and the new line inserted with the following:

I <ret>
<tab> OUT <tab> 1BH <ret>
<CTL-Z>

Another approach would be to move the CP 15 spaces to the right with the command below:

15C <ret>

This move places it just before the C in the line. The C can be removed with the delete command:

D <ret>

Then, the correct text can be inserted with the insert command:

I <ret>
1BH <ret>
<CTL-Z>

Now that the corrections have been made, the new, corrected file should be saved on disk. This can be done by typing the following:

E <ret>

The edit session ends with this, and control is returned to

the operating system. The corrected file is called "DIAG.ASM". A backup file called "DIAG.BAK", is created by ED; it contains the old version of DIAG.ASM. Thus, if the corrections were wrong, the old version could be retrieved and renamed DIAG.ASM.

This example introduces the major commands in ED. A more complete list of commands is given in Table 2. A number (symbolized by ±n) can proceed most of the commands to cause them to repeat. If the number is missing, +1 is assumed. Thus, T types one line, while 20T types 20 lines. The pounds sign (#) can be used in place of a number to mean "all"; therefore, #T types all lines (after the current CP position).

The operator manual for ED should be consulted for a complete description of all the commands in ED.

TABLE 2. ED COMMANDS.

nA<cr>	–	Append the next n unprocessed source lines from the source file at SP to the end of the memory at MP. Increment SP and MP by n.
±B<cr>	–	Move CP to beginning of memory if +, and to bottom if -.
±nC<cr>	–	Move CP by ±n characters (toward front of memory if +), counting the <cr><lf> as two distinct characters.
±nD<cr>	–	Delete n characters ahead of CP if plus, and behind CP if minus.
E<cr>	–	End the edit. Copy all buffered text to temporary file, and copy all unprocessed source lines to the temporary file.
±nK<cr>	–	Kill (ie remove) ±n lines of source text, using CP as the current reference. If CP is not at the beginning of the current line when K is issued, then the characters <u>before</u> CP remain if + is specified; while the characters <u>after</u> CP remain if - is given in the commnad.
±nL<cr>	–	If n=0, then move CP to the beginning of the current line (if it is not already there); if n≠0, then first move the CP to the beginning of the current line, and then move it to the beginning of the line which is n lines down (if +), or up (if -). The CP will stop at the top or bottom of the memory if too large a value of n is specified.
Q<cr>	–	Quit edit with <u>no</u> file alterations. Return to CP/M.
±nT<cr>	–	If n=0, then type the contents of the current line up to CP; if n=1, then type the contents of the current line from CP to the end of the line. If n>1, then type the current line along with n-1 lines which follow – if + is specified. Similarly, if n>1, and - is given, type the previous n lines, up to CP. The break key can be depressed to abort long type-outs.
±n<cr>	–	Equivalent to ±nLT, which moves up or down and types a single line.

EXAMPLE B: EDITING.	
Given:	The file ER.ASC, which contains the following: ENERGY CONSERVATION IS THE MORAL EGIVALENT OF WAR.
Find:	The commands required to correct the spelling error and save the corrected text.
Solution:	Evoke the editor from the operating system: ED ER.ASC <ret> Bring the file into memory: #A <ret> Position CP at the beginning: B <ret> Move to the start of the next line: L <ret> Move forward 7 characters: 7C <ret> Check that G is the next by typing the line from the CP: T <ret> Delete G: D <ret> Insert the missing QU: I <ret> QU <CTL-Z> Save the corrections and leave ED: E <ret>

PROGRAM ASSEMBLY

To execute a program written in assembly language, the operator must convert it first to machine code. This is done with the assembler called "ASM" and the loader called "LOAD". An example based on the DIAG.ASM program is sufficient to illustrate their use.

Assembly is initiated by typing ASM, then typing the name of the file. The filetype should not be supplied, since the assembler requires that the type be ASM. To assemble DIAG, type the following response to a system prompt:

ASM DIAG

The editor will then take over. If there are no errors, the machine code result will be stored on a disk as a file under the name DIAG.HEX. When assembly is complete, the assembler returns control to the operating system.

It is difficult to view the results of the assembly. Typing DIAG.HEX would not make sense because the TYPE command requires ASCII code, and DIAG.HEX is in hexadecimal code.

To simplify viewing the results of assembly, ASM generates a printable file called DIAG.PRN; when displayed, using TYPE, this shows both the assembly code and the assembled machine code.

If errors are encountered in assembly, error messages will be displayed and the output files may not be generated; this depends on the type of error.

PROGRAM EXECUTION

Any program in machine code should have a filetype with the name COM. The assembly of DIAG created a HEX filetype.

The system has a program called LOAD that creates a COM file from a HEX file. The command is as follows:

```
LOAD DIAG
```

This will generate a file called DIAG.COM that can be run.

When a so-called COM file is on a disk, it can be run, or executed, by simply typing its name without a filetype. So, to execute DIAG, which draws a diagonal, type the following:

```
DIAG
```

The computer executes the program, and then returns to the operating system.

The complete commands needed to assemble and run DIAG are listed below. This example assumes a correct assembly file exists with a filename DIAG.ASM. The user types the following:

```
ASM DIAG  <ret>
LOAD DIAG <ret>
DIAG      <ret>
```

This program creates the files DIAG.PRN, DIAG.HEX, and DIAG.COM. To run the program again, only DIAG needs to be typed, since DIAG.COM exists. The ASM, PRN and HEX versions of DIAG could be erased with ERA without affecting the subsequent execution of DIAG.

BASIC

BASIC is a language that makes programming easier by having the computer do much of the tedious parts of programming. For instance, the effect of DIAG can be accomplished with the BASIC program that follows:

```
10 PEN UP
20 JUMP TO -64,51
30 PEN DOWN
40 JUMP TO 38,-51
```

The meaning of the instructions will be studied in future modules. The purpose of this example is to introduce a program that is shorter and more like English than the assembly version.

There are two types of BASIC: compiled and interpreted. When BASIC is compiled, programs like the one above that use BASIC commands are converted to machine code in much the same way an assembler converts assembly code. The process of converting a high level language like BASIC to machine code is called compilation. The program that accomplishes the compilation is called a complier. A compiler is a program that accepts a file composed of BASIC commands and generates a file that can be loaded and run. A BASIC compiler, called BASIC-E, will be used in the laboratory.

An interpreted BASIC is easier to use because there is no compilation required. To run an interpreted BASIC program, the file BASIC/5 is executed by typing BASIC/5 in response to the system prompt; then, programs can be run, entered, saved, edited, and removed under control of that program. The following two modules will detail how this is done.

One of the commands within BASIC/5 is RUN. When this command is typed, each line of BASIC code is examined by the BASIC/5 program and converted to machine instructions as the program is executed. This process is called interpretation. In a sense, when an interpreted BASIC is run, compilation and execution are combined.

Interpretation can be a very inefficient form of compilation. If the program contains a loop that results in one BASIC instruction being executed many times, an interpreted BASIC will reinterpret the same line every time through the loop. As a result, interpreted BASIC is slow.

The advantage of interpreted BASIC is that programs are easy to modify. To add a line of BASIC code, the line is simply typed in and the program RUN again. Contrast this to the more cumbersome process in a compiled BASIC: The original file must be changed with ED; then the source file must be compiled again with BASIC-E; and then the program can be loaded and run.

EXERCISES

1. State the commands that would be required to generate a file that could be typed out to give the following:

```
PARIS
IN  THE
THE SPRING
```

2. Suppose the file in Exercise 1 resides on disk. Give all the commands required to edit that file so that, when typed, it would give the following correction:

   PARIS IN THE SPRING!

3. Describe the steps required to enter and execute a file that contained the following BASIC program:

```
10 FOR I=1 to 10
20 PRINT 2*I+1
30 NEXT I
40 END
```

4. What are the differences among PRN files, HEX files, and ASM files?
5. What are the relative advantages of compiled and interpreted BASIC?
6. Describe the steps required to enter, correct assemble, list, and execute a program written in assembly language.
7. Describe how one would enter and execute a listing of a program in machine code.

LABORATORY MATERIALS

Access to a disk-based microcomputer with CP/M, BASIC/5, and BASIC-E.

1 floppy disk 8 1/4".

LABORATORY PROCEDURES

LABORATORY 1: FILE MANIPULATION.

1. Create a personal disk.

   A master disk will be available for each student to use to copy onto a disk that he or she will be loaned for the duration of this course. Copy the operating system onto the disk, using SYSGEN. Then use PIP to copy the editor (ED.COM), assembler (ASM.COM), loader (LOAD.COM), interactive BASIC (BASIC.COM), and compilation instructions (COMP.TYPE). Place the disk in drive A and reset the computer by pressing the reset button. If the disk has the operating system copied correctly, the computer should respond with a system prompt. Use DIR to confirm that everything is on the disk.

2. Create DIAG.

   Enter the program DIAG exactly as shown in Table 1. Save it and leave the editor. Type out the file, and call up ED again to correct line 5 and any other errors that may have been made. Save the corrected file, and do not erase the backup version.

3. Execute DIAG.

   Assemble, load, and execute DIAG. Record the image seen on the TV screen in Data Table 1 (Files). Use DIR to list the complete names of all files with filename DIAG and any type name. Record these file names, type out and note their contents, and describe when each was

created in Data Table 1 (Files).

LABORATORY 2: EXECUTING BASIC.

1. Run an interpreted BASIC program.

The program in Table 3 is a nonsense program that performs ten thousand additions and multiplications by requiring line 40 to be executed that many times.

TABLE 3. A BASIC PROGRAM.

```
10  FOR I = 1 TO 100
20  S = I
30  FOR J = 1 TO 100
40  S = S+J*.001
50  NEXT J
60  NEXT I
70  PRINT "THE SUM IS", S
```

The logic and meaning of the program is unimportant; it simply will be used for practice and for comparison of the two forms of BASIC.

Note the time. Execute the interpreted BASIC by typing its name, BASIC/5. In response to its questions, tell it to create a new program named "COMPUTE." Then enter the program given in Table 3. Check for errors by typing the following:

LIST <ret>

Correct errors by retyping the incorrect line.

When correct, execute the program by typing RUN. Record (in Data Table 2) how long it took to get the program into the computer and running, and also how long it takes to run. It is finished running when it types

the following:

THE SUM IS _____

READY

Record what it reports as the sum in Data Table 2 (BASIC).

Save the program by typing the command:

SAVE

Return the system by typing the command:

SYSTEM

2. Modify interpreted BASIC.

Suppose that line 40 of the program should be the following:

40 S = S+I/J

To do this, recall BASIC/5 with the old program COMPUTE by typing the last line:

BASIC/5 OLD COMPUTE <ret>

Note the time. Now type in the corrected line, and check the listing of the program for errors and run it.

How long did it take to correct the program? How long did it take to run it? What number did the program report as the sum? Record answers in Data Table 2 (BASIC).

3. Run a compiled BASIC.

The disk contains detailed instructions for compiling BASIC. Read the instructions by typing the file called "COMP.TYPE". Follow those directions to enter and run the program in Table 3.

Again, note the time required to get the program to the point that it will run, record its execution time, and copy the sum that the program computes in Data Table 2 (BASIC).

4. Modify compiled BASIC.

Make the correction to line 40 described in Step 2

and run the program. This must be done using ED to modify the source file and then compiling, loading, and executing the program. Record the time required to obtain a modified COM file, the execution time, and the results of the computation in Data Table 2 (BASIC).

DATA TABLES

DATA TABLE 1: FILES.

STEP 3: Describe results of running DIAG _____

_____

_____

DIAG FILES

Extent	When Created	Use

DATA TABLE 2: BASIC.

STEP	Time to Enter	Time to Execute	Sum
1			
2			
3			
4			

Describe the relative merits of interpretation and compilation: _____

_____

_____

REFERENCES

An Introduction to CP/M Features and Facilities. Digital Research Corp.

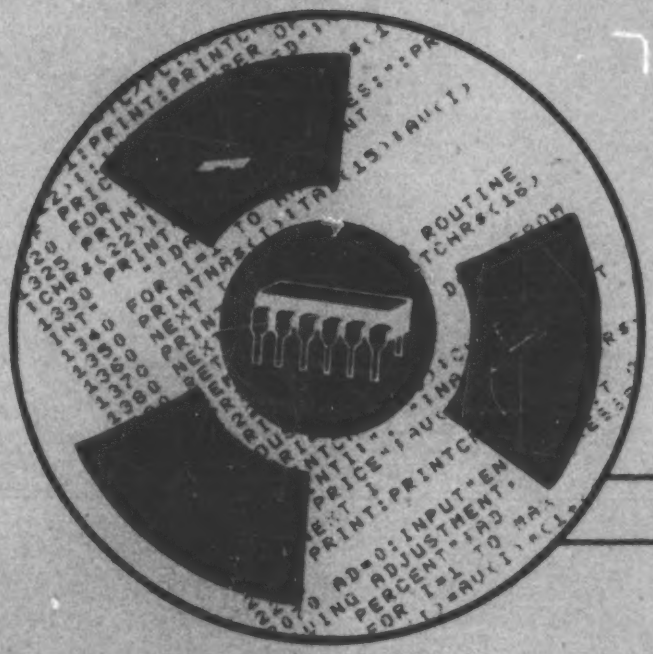
ED: A Context Editor for the CP/M Disk System. Digital Research Corp.

CP/M Assembler (ASM) User's Guide. Digital Research Corp.

ENERGY TECHNOLOGY

CONSERVATION AND USE

ecut



MICROCOMPUTER OPERATIONS

MODULE MO-05

ENERGY APPLICATIONS OF MICROCOMPUTERS

CORD CENTER FOR OCCUPATIONAL RESEARCH AND DEVELOPMENT

INTRODUCTION

The versatility of a microcomputer can be illustrated by operating two application programs related to energy conservation. The first is a solar energy feasibility study that uses field data to estimate the economic value of solar heating. Students apply this data to local conditions.

The second program illustrates load shedding – a strategy used by large consumers of electricity to keep their peak demand under control. In this module, load shedding becomes a game in which the student has the problem of trying to find a shedding strategy that conserves the most energy with the minimum of inconvenience. Two versions are provided: In one, the student does the shedding; in the other, the student programs the computer-controlled shedding. The results are compared to demonstrate the computer's ability to make quick, accurate decisions.

PREREQUISITES

The student should have completed Modules MO-01 through MO-04 of Microcomputer Operations.

OBJECTIVES

Upon completion of this module, the student should be able to:

1. Describe a solar energy heating feasibility study, the data such a study requires, and the kinds of calculations this implies.

2. Describe the significance of load shedding, how it is accomplished, what kind of data it requires, and what calculations are required.
3. Estimate the hardware and software requirements of a computer that could perform solar energy feasibility studies.
4. Execute programs that perform a solar energy feasibility study and that emulate a load shedding problem.

ENERGY CONSERVATION PROGRAMS

This module provides two concrete examples of moderately complex BASIC programs that can be run on a small microcomputer. To use and appreciate these programs, the student needs a brief introduction to the ideas behind the programs, which is the primary objective of the following sections of this module. The programs will be run in the laboratory and then analyzed in terms of the computer hardware and software they require.

The first program is SOLAR. This is a program that determines whether a particular solar installation can save money in the long run. The second program simulates load shedding – a procedure used by commercial electricity consumers to conserve electricity and reduce costs.

SOLAR

Solar collector systems are gaining in popularity as a means to generate space heating and hot water. However, there are still questions that arise concerning the value of solar heating. Opponents say it is too expensive; advocates claim that it saves fuel and money. It is a fact that certain solar systems can be economical under certain circumstances; but, the economic viability of a solar installation depends on its design and location, on present and future costs of fuel, and on interest and tax rates.

SOLAR is a BASIC program that can perform the calculations necessary to determine the economics of particular solar installations. This program is a good example of the

type of computations that a small computer can perform; and it will be run as part of the laboratory work to be done in this module.

SOLAR assumes the system diagrammed in Figure 1 is used. The collector heats air and stores it in a thermal reservoir, which normally can store enough heat for a few days of heating. Heat is then pumped from the reservoir to the building. If necessary, a backup heater can be used to supplement the reservoir or take the heat up to an acceptable level.

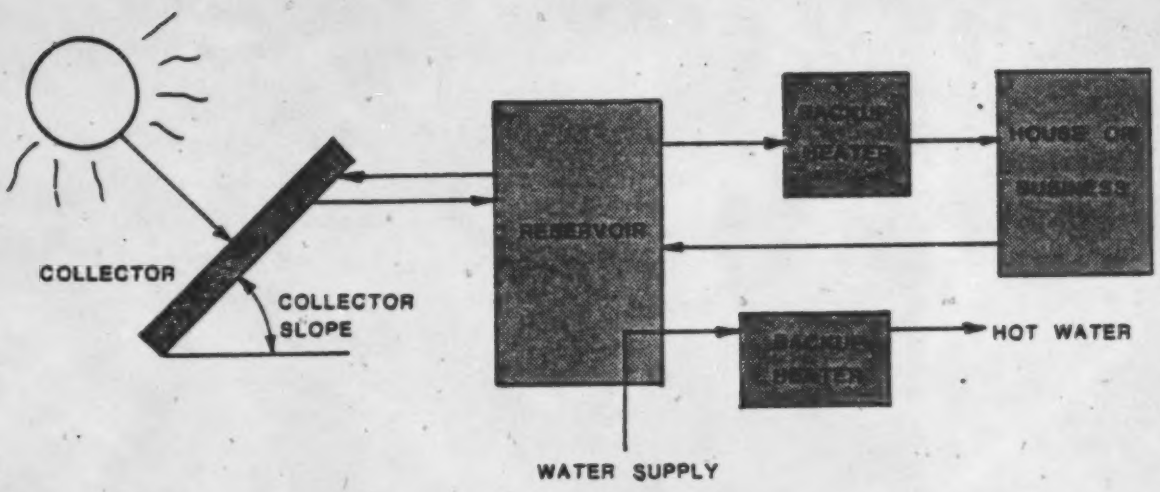


Figure 1. Solar Heating System Used by SOLAR.

The Collection System

The collector is assumed to face south, and its slope is a particularly critical design parameter. For best year-round performance, the slope should equal the latitude. Better winter performance can be obtained with a 10°-20° greater slope.

The collector area is a critical cost factor. Collectors that circulate a liquid cost approximately $100 per square

meter. Air collectors cost less – in the range of $30 per square meter. For simplicity, SOLAR assumes that air collectors are used. Large collectors are needed to collect heat for extended storage because long-term storage is needed for sunless days. However, it is usually more economical to have storage for only a few days and then rely on backup heating – the alternative requires the use of prohibitively large collectors and reservoirs.

Air systems use bins that are filled with stone for reservoirs. A good rule of thumb is to use 0.1 to 0.4 cubic meters of stone for every square meter of solar collector.

Many different solar collectors are available. An ideal collector would collect all the available solar radiation and convert it to useful heat without losing any heat by convection. The performance of any collector can be described with two numbers:

- The value of E – the conversion efficiency, the fraction of the incident solar energy that actually gets into the heating system (typical values are 0.5 to 0.8).
- The value of L – the energy lost from the collector, per square meter of collector, per degree temperature difference between the collector and the outside (typical values are 3 to 5 watts per square meter, per degree Celsius).

An ideal collector would have E = 1 and L = 0.

Heat Loads

The heating system is used for both space heating and hot water. The energy required for space heating depends on

climate and the overall heat-loss from the structure. Climate information is summarized by the number of degree-days a location has each month. Degree-days are the monthly sum of average daily inside-outside temperature differences. The computer program has these data for ten selected U. S. cities.

The overall heat-loss for a structure depends on its size, as well as how tight and well-insulated it is. The heat-loss is expressed as a rate of energy-loss (in watts) per degree difference between inside and outside (in Celsius). Values from a few hundred (for a small tight house) to thousands (for a large, drafty house) to tens of thousands for industrial structures are possible. The laboratory section gives the means of estimating this heat-loss by taking certain information from heating bills.

The heat required by hot water depends on how much water is used and how much it must be heated. An average family uses 100 liters per day per person. The water must be heated to 55°-77°C from the temperature of the water supply, which is typically 5°-15°C.

Economic Factors

The problem with comparing the relative costs of solar and conventional heating is that the costs are incurred at different times. Solar systems are expensive to install, but result in steady savings over their lifetime. Conventional heating is less expensive to install but will become increasingly expensive to use as fuel prices rise.

Economists resolve this sort of problem by posing the following question: "If a large sum of money is borrowed, would it be better in the long run to invest it and pay conventional fuel costs, or to use the money to buy a solar system?"

The answer depends on tax rates, fuel costs, inflation rates, interest rates, and investment opportunities – both now and in the future.

When given all these numbers (answers), the student should make a sound, economic comparison of solar and conventional heating using SOLAR to perform the calculations. However, it is very difficult to estimate the exact value of these numbers for as long as 20 years into the future, but one can try out a range of values and see how these changes affect the calculations.

The final result of SOLAR's economic analysis is called the "present worth of the total solar savings." If this number is positive, it represents today's value of the net solar savings. It is less than the actual dollar savings because a savings in the future is not worth as much as an equivalent savings today. For example, a savings of $48.30 today is equivalent to saving $100 in 10 years at an 8% annual interest rate.

If the present worth of the total solar savings is negative, then solar heating is more expensive. Detailed instructions for determining the savings potential for structures will be given in the laboratory.

LOAD SHEDDING

Electric utilities must have sufficient electricity generation capacity on hand to meet the largest, or peak, load. Peak loads usually occur on early afternoons in the summer; during the remainder of the time, electric utilities have unused capacity, as illustrated in Figure 2.

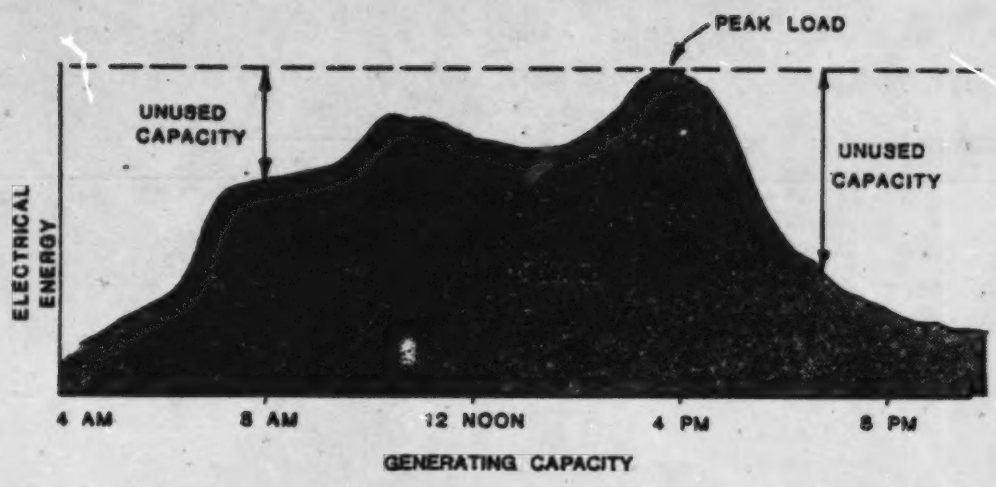


Figure 2. Typical Electricity Demand in Summer.

Utilities must build power plants to meet peak loads even though the average load might be much less. If they can limit their peak load, they can save expensive construction and conserve fuel. One way to do this is to give an economic incentive to users to limit their own peak demand.

To limit the load, discounts are sometimes given to users who keep their peak load below some maximum value. Users must be able to shut off some part of their load to keep within the maximum – this is called load shedding.

To see how load shedding might work, consider a company that has 10 electric hot water-heaters. Each heater comes on when the water in its tank is too cool. The heaters all work independently, so it is possible that all ten might come on at once. This would create a huge peak load, which is unnecessary, since some of the heaters could be left off while others are on. By staggering the heaters, the maximum load could be kept much lower.

Load shedding is an ideal application for a small microcomputer system. The computer can be programmed to make the appropriate choices to keep the total load under a pre-set maximum, while still delivering electricity where it is most needed.

Both load shedding and the use of a microcomputer to control the shedding are illustrated in programs that will be run in the laboratory. These programs simulate the electrical needs of a small company. Two versions of the program will be run. The program MANUAL simulates the electrical needs of the company, but has the student control the shedding. The program AUTO simulates the same company and automatically sheds the load. Both simulate a company with the 10 major electrical loads listed in Table 1.

TABLE 1. COMPANY ELECTRICAL LOADS.

Number	Load Type	Load Size	Target	Penalty
1	Heater	2	Exchange between 50° & 90°C	0
2	Heating pump 1	8	Area 1 between 25° & 29°C	5
3	Heating pump 2	6	Area 2 between 25° & 29°C	2
4	Water heater 1	15	Water 1 between 40° & 60°C	3
5	Water heater 2	10	Water 2 between 40° & 60°C	1
6	Air compressor	10	Pressure between 4 and 8 atmosphere	4
7	Ice melter	20	Pavement between 0° & 10°C	2
8	Exterior lighting	12	On after 4:00 p.m.	1
9	Water pump	4	Reservoir between 1000 & 5000 gal	3
10	All other	0-30 variable		10

Each load carries a different electrical demand. The size of each load is listed in the "Load Size" column. These units are arbitrary.

Figure 3 illustrates the company heating system. Load 1 represents the electrical power required to fire an oil heater and pump hot water out of it into a heat exchanger. Load 2 is the power needed to pump water to heat the offices in the company called "Area 1." The circulating heating water is warmed in the heat exchanger and cooled in the offices. Load 3 is the power needed to pump water to heat the production area called "Area 2."

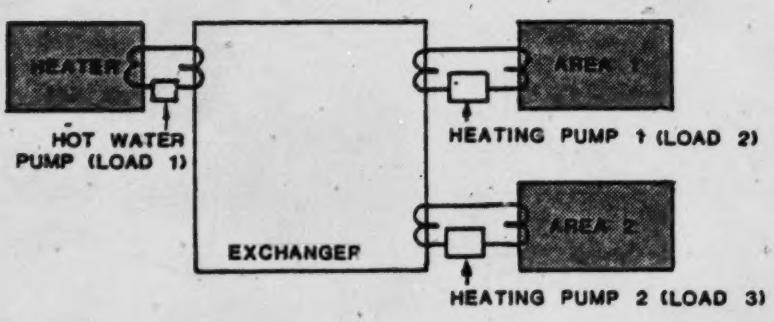


Figure 3. Simplified Company Heating System.

Load 4 is an electric hot-water heater for Area 1. Load 5 is a second hot-water heater for Area 2. Load 6 is a large air compressor used in production.

Because it is snowing outside, an electric ice melter buried in the pavement outside the door (Load 7) is needed to keep the pavement from freezing.

After 4:00 p.m., the company's exterior lights turn on (Load 8). The company has a water pump (Load 9) that is used to fill a roof-top, 5000-gallon reservoir.

Finally, there are numerous motors, lights, and utilities that are necessary, which are lumped together as Load 10. This load is variable; being almost zero after hours and reaching as high as 30 during the day, but dipping significantly during lunch between 12:00 (noon) and 1:00 p.m.

The easiest way to shed load would be to turn everything off. This, of course, is impractical. There is a target range for each load, however. For instance, Load 2 controls Area 1's temperature, which should range between 25° and 29°C. If the temperature drops below this range, management gets upset and the controller receives penalty points. The number of points accumulated per minute outside of the target range is listed in the last column shown in Table 1.

Note that there is no direct penalty for failing to keep the heat exchange temperature high enough. However, if the heat exchanger is not hot, neither Area 1 nor Area 2 can be heated when their pumps are turned on.

The load shedder must take into account the various penalties. If it is necessary to shed a load that might be needed to keep on target, it is best to choose the load with the smallest penalty.

The programs simulate 10 hours of a typical winter day for the company in question. The maximum load is set at 50 units. The program MANUAL stops after simulations each 10 minutes and asks the user whether any loads should be turned on or off. After the changes are made, another 10 minutes is simulated. The total load is shown along with the percentage of the maximum permitted. Total energy used and

penalty units given are accumulated and displayed. If the user starts one 10-minute interval at a load above the maximum allowed, the user is fired and the program halts. If random variables raise the energy usage above maximum during one interval, there is no penalty as long as the overload condition is corrected for the next interval.

Detailed instructions for running this program are given in the laboratory section.

LABORATORY MATERIALS

Access to a disk-based microcomputer with CP/M and BASIC/5.
1 floppy disk 8 1/4" containing SOLAR, MANUAL, and AUTO.

LABORATORY PROCEDURES

LABORATORY 1: SOLAR.

GATHERING THE STATISTICS

The purpose of this laboratory is to perform a realistic economic analysis of a typical solar installation in a home. To do this, data must be gathered about the house to be heated, the solar system being installed, the cost of fuel, and other economic factors. The data required are described below. Once collected, it should be recorded in Data Table 1 (SOLAR).

1. Choose a house for the calculation.

   The hot-water load is determined by the number of people living in the house. The space-heating load can be estimated from a typical monthly heating bill, the number of degree days during that month, and knowledge of the type of heating.

2. Find the space-heat load.

   SOLAR needs the variable U, which is the overall energy-loss coefficient area product. U can be found from an existing house, using the following equation:

$$U = \frac{FHe}{D} \qquad \text{Equation 1}$$

where:

F = The amount of fuel consumed during one month.
H = The heating value of the fuel in the same units.
e = The average efficiency of the furnace.
D = The number of degree days during the month chosen.

F can be found on the heating bill and H and e can be found in Table 2. For gas and oil, e can be adjusted up 5% for burners in good repair or down 5% for burners in poor repair. To find D (number of degree days) for the same month that was chosen for the information from the fuel bill, call any fuel oil distributor or a government weather service. For a new house, U will have to be found from an energy audit. (See ASHRAE Handbook, Chapter 21.)

TABLE 2. VALUES OF H AND e FOR DIFFERENT FUELS.
(MJ is a megajoule or 1 million joules of energy.)

Type of Heating	Value of H in SI Units	Value of H in British Units	Value of e
Natural gas	41 MJ/m^3	1.1 MJ/ft^3	0.55
Fuel oil	39 MJ/liter	140 MJ/gal	0.55
Electricity	3.6 MJ/kWh		1.0

3. Design the size and orientation of the solar collectors. The size and orientation of the solar collectors are the most important choices to be made. First, look at the house and determine how the panels could be mounted on the south-facing side. It is best if the collector slope is about 15 degrees more than the latitude. Measure the

slope that would be used and measure how many degrees away from due south the collectors would face. This is called the azimuth angle; it should be positive for west-facing collectors and negative for east-facing collectors.

The collector size is an important factor; in square meters, it should be approximately one-third of U. It is advisable to repeat all the calculations for several different collector sizes to find the most economical size. Estimate the largest size the house could accommodate.

5. Choose the collectors.

Three numbers are needed for the collectors:

- The value of E – the conversion efficiency.
- The value of L – the energy loss coefficient.
- The value of V – the variable cost in dollars per square meter of collector.

Typical values for these are E = 0.7, L = 4 $W/m^2C°$, and V = $\$30/m^2$.

A local business that installs solar collectors could give some help on the figures needed. The definitions of E and L, in terms that might be familiar to solar technicians, are as follows:

$$E = F_R(\tau\alpha)_n$$

$$L = F_R U_L$$

Equation 2

where:

F_R = The collector's heat removal efficiency factor.
τ = The solar transmittance of the transparent covers.
α = The solar absorption of the collector plate.
U_L = The collector's overall energy-loss coefficient.

NOTE: See references for further details (e.g., "Solar Heating Design," Beckman, et al, Chapter 2.)

5. Design the solar system.

SOLAR needs the storage capacity, S, of the reservoir and the total cost, F, of the heating system – except for the collectors. The storage capacity in cubic meters should be about one-fourth of the collector area in square meters. Use this with the help of a business that sells solar equipment to estimate F for the house in question. Assume the current heater will be used for backup, so do not include its cost in F, even if it needs replacement. Typical values for F are $1,000 to $2,000.

6. Finance the solar system.

The solar system is assumed to be financed by a mortgage. SOLAR needs to know the percentage of down payment required, the interest rate, and the length of the mortgage.

7. Other parameters.

SOLAR needs estimates of the following:

a. Income tax rate – the average percentage of income paid in taxes by the home occupants.
b. Property tax – the actual percentage of the property value paid in taxes each year.
c. Property tax change – the annual percentage rate of increase of property tax (check with the local collector of taxes).
d. The cost of fuel – how much fuel costs per unit

of energy. SOLAR needs this in cost per megajoule. A fuel bill will give fuel costs in cost per unit; then Table 2 can be used to convert this cost.

e. The rate of increase of fuel costs (percentage per year) – this number will be difficult to estimate over twenty years. Between 1978 and 1979, oil increased 100% in New England; but that rate is unlikely to continue. Check with the local utility or fuel supplier for information concerning increases over the last five years.

f. The salvage value – SOLAR needs an estimate of the value of the solar system at the end of the analysis period (twenty years). The estimate is in terms of a percentage of the cost of the system. If it will have to be totally replaced, the value is zero; if it requires no change, the salvage value is 100%.

RUNNING THE PROGRAM

8. Run SOLAR.

Load BASIC and request the program, SOLAR. When it is run, it will interactively request values for all the parameters previously described. Enter those values and record the results.

9. Find SOLAR's memory needs.

When SOLAR runs, the operating system is in high-numbered memory locations; the BASIC interpreter is in low-numbered memory locations; and the program is

between these boundaries, as shown in Figure 4. The addresses can be found in the BASIC command, "GETADDR(N)," where N can be 1, 2 or 3, as shown in Figure 4. The only unknown is how much memory is occupied by variables. Five bytes are required for each variable used in the program.

Use GETADDR to find the memory used by SOLAR and BASIC. Given the information that 300 variables are used, estimate the smallest amount of memory required by SOLAR.

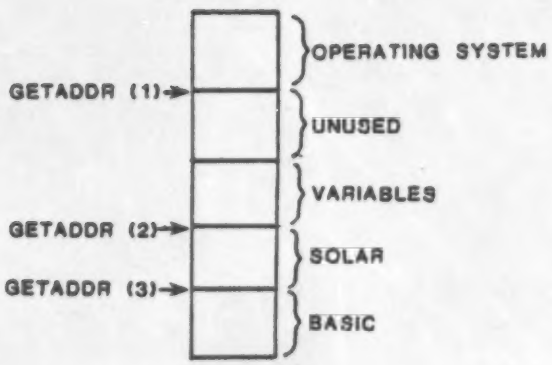


Figure 4. Memory Map.

LABORATORY 2: LOAD SHEDDING.

1. Run MANUAL.

Load the BASIC program MANUAL and run it. It will display the status of electrical use similar to that shown in Table 3. At the end of each simulated ten minutes, the program will ask whether any loads should be turned on or

off. By typing the number of the load, its status will be changed (from on to off or off to on). If a mistake is made, type the number again to restore its status. When all changes have been made, type "0" (zero) to instruct the computer to simulate another ten minutes.

The program stops at the end of the day. Record the total energy consumed and the penalty units incurred in Data Table 2 (Load Shedding). If time permits, repeat the program to get more experience.

TABLE 3. TYPICAL DISPLAY FOR LOAD-SHEDDING PROGRAMS.

Number	Load	Load Size	Penalty	Controls	Target	Value	Status (1=ON, 0=OFF)
1	Heater	2	-	Exchange Temp.	50-90	60	1
2	Heating Pump 1	8	5	Area 1 Temp.	25-29	29	0
3	Heating Pump 2	6	2	Area 2 Temp.	25-29	25	1
4	Water Heater 1	15	3	Area 1 Water Temp.	40-60	45	0
5	Water Heater 2	10	1	Area 2 Water Temp.	40-60	42	1
6	Air Compressor	10	4	Tank pressure	4-8	3	1
7	Ice Melter	20	2	Pavement Temp.	0-10	2	0
8	Exterior Lights	12	1	Outside Lights	On after 4	3417	1
9	Water Pump	4	3	Gallons in tank	1000-5000		0
10	All other	8	10				1

2. Run AUTO.

Load and run the BASIC program AUTO. This program generates a display similar to MANUAL, but it makes the load-shedding choices automatically. Record the results of a simulated day's run, using AUTOMATIC, in Data Table 2 (Load Shedding). Does it do as well as it does in MANUAL?

3. Determine the memory required by MANUAL and AUTO.

Follow the procedures taken in Step 9 of the previous laboratory for both load-shedding programs. A major part of these programs consists of simulating the situation. The part of AUTO used to control the load can be estimated as the difference in length between MANUAL and AUTO. This difference, plus BASIC, is what would be required for an actual load-shedding computer. Record this value in Data Table 2 (Load Shedding).

DATA TABLE 1: SOLAR.

STEP 1: Number of people in house: _____

STEP 2: Method used to find U: _____

_____

_____

Calculations: _____

_____

Value of U: _____

STEP 3: Collector slope: _____

Azimuth angle: _____

Largest possible collector size: _____ m^2

Collector size chosen: _____ m^2

STEP 4: Coversion efficiency: E _____

Energy loss coefficient: L _____ $W/m^2C°$

Area cost of collector: V _____ $/m^2$

Describe how obtained: _____

_____

_____

_____

STEP 5: Storage capacity: S _____ m^3

Total fixed cost: F _____ $

STEP 6: Down payment: _____%

Interest rate: _____% per year

STEP 7: Tax rate: _____%

Yearly Property tax rate: _____ %

Yearly Property tax change: _____ %

Cost of fuel: _____ ¢/MJ

Annual fuel cost increase: _____ %

Salvage value: _____ %

Data Table 1. Continued.

STEP 8: Attach print-out if possible. Otherwise, record the present worth of solar savings for each year and total over 20 years.

Year	Saving	Year	Saving
1		11	
2		12	
3		13	
4		14	
5		15	
6		16	
7		17	
8		18	
9		19	
10		20	

Total Solar Saving: _____

STEP 9: GETADDR(1): _____
GETADDR(2): _____
GETADDR(3): _____
Memory required by BASIC: _____
by SOLAR: _____
(include variable storage)

DATA TABLE 2: LOAD SHEDDING.

STEP 1: MANUAL

Energy consumed: _____

Penalty units: _____

STEP 2: AUTO

Energy consumed: _____

Penalty units: _____

STEP 3:	Manual	Auto
GETADDR(1):	_____	_____
GETADDR(2):	_____	_____
GETADDR(3):	_____	_____

Memory required by BASIC: _____

by MANUAL: _____

by AUTO: _____

Approximate memory required by a load-shedding program: _____

REFERENCES

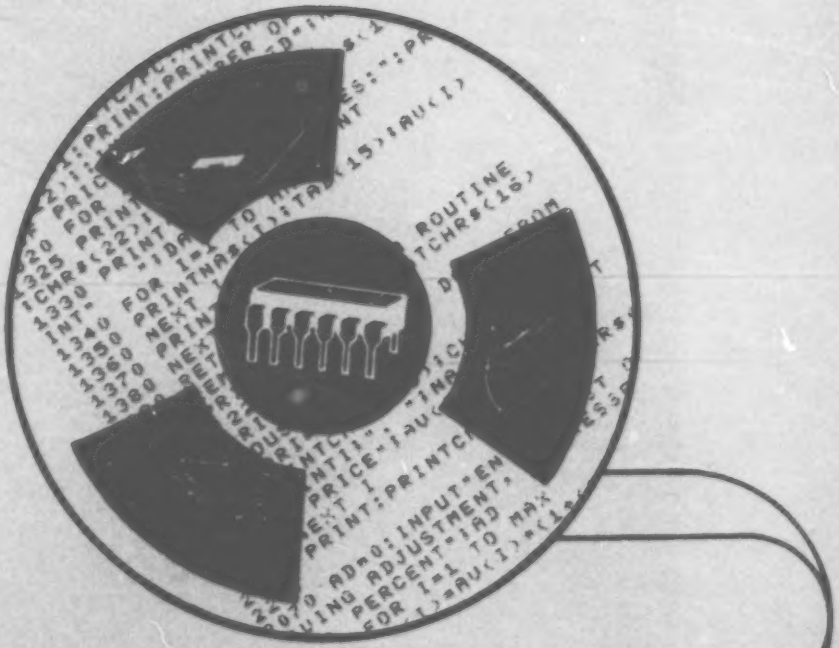
Beckman, et al. Solar Heating Design. Wiley Interscience, 1977.

ASHRAE. Handbook of Fundamentals. American Society of Heating, Refrigerating, and Air Conditioning Engineers, 1972.

ENERGY TECHNOLOGY

CONSERVATION AND USE

ecut



MICROCOMPUTER OPERATIONS

MODULE MO-06

INTRODUCTION TO BASIC

ORD CENTER FOR OCCUPATIONAL RESEARCH AND DEVELOPMENT

INTRODUCTION

Computers "speak" but one language – machine language – and all instructions given to the computer must ultimately be in this form. However, to program the computer directly in machine language is unnecessary. Not everyone possesses this skill, which takes a considerable time to develop. The average person can communicate with a computer via a number of higher-level languages that are more like English. These languages are read into the computer's memory, and from that point on, the operator can use a language that is easier, while the computer translates that to machine language – a process called "interpreting." Of course, the more sophisticated the language, the more memory is required for interpreting, which is one reason there is no one language that is universal or unlimited in application.

To load a computer's memory bank with the entire unabridged dictionary would not be economically feasible when only a handful of words is necessary. For instance, in the scientific and engineering community, FORTRAN (FORmula TRANslator) is a practical language to use; it is particularly designed to process formulas and scientific terminology. COBOL (COmmon Business-Oriented Language) is used in most business and accounting applications.

In this module, the elements of a simple, all-purpose – but limited – language called BASIC will be presented. BASIC is an abbreviation for Beginner's All-purpose Symbolic Instruction Code. It enables the user, for instance, to enter simple codes such as "+", which will cause the computer to execute the machine language program that adds two numbers together.

An important point that should be understood concerning the language of BASIC is that, while it is widely used, there are also many other variations in use. Some versions of BASIC have a limited vocabulary and will not recognize commands that are part of other BASIC programs. Students must become familiar with the particular version of BASIC for the computer that they have access to.

PREREQUISITES

The student should have completed Module MO-01 through MO-05 of Microcomputer Operations.

OBJECTIVES

Upon completion of this module, the student should be able to:

1. Define the action of the following commands in graphical BASIC:
   a. ERASE
   b. SCROI L ON
   c. SCROLL OFF
   d. MOVE N
   e. TURN N
   f. HOME
   g. DOT
   h. PEN UP
   i. PEN DOWN
   j. ERASER ON
   k. ERASER OFF

l. HOP N
m. JUMP TO X,Y
n. LET
o. INPUT
p. GOTO
q. FOR
r. NEXT
s. NAME
t. NAME P
u. SAVE
v. OLD
w. RUN
x. LIST
y. PRINT

2. Determine the action caused by programs using these commands.
3. Write short programs using these commands.
4. Define the following terms:
   a. Graphics commands.
   b. Scrolling.
   c. Screen coordinates.
   d. Variable.
   e. Expression.
   f. Floating point.
   g. Mantissa.
   h. Loop.
   i. Iteration.
   j. Range.
   k. Rules of precedence.
   l. Immediate mode.
   m. Programmed mode.
5. Evaluate BASIC expressions, using the rules of precedence of the operators +, -, and /.

6. Evaluate numbers in floating point notation.
7. Use BASIC as a calculator to evaluate expressions.

BASIC GRAPHICS

BASIC is a widely-used language that has many practical applications. It is especially useful to the beginning student as a "first" language to learn in computer programming. The language consists of a series of instructions that are introduced in logical groups and then applied to typical programming problems.

Not all BASICs are the same. The particular version introduced in this module includes the ability to draw figures and designs on a TV display. This ability makes it easier to learn the language but is not part of most BASICs.

In the sections that follow, no distinction will be made between the words "command" and "instruction." There are situations in computer operation where these words do have different connotations.

CORE GRAPHICS

The first set of instructions to be studied in this module are the six listed in Table 1, which control the graphical display on the video screen. These instructions, called graphic commands, are unique to the BASIC language used in this module because many computers which have BASIC capability do not necessarily have video terminals that have graphic capability.

TABLE 1. CORE GRAPHICS COMMANDS.

COMMAND FORM	ACTION
ERASE	Erases the entire screen.
SCROLL ON	Permits later RETURNs to scroll the screen.
SCROLL OFF	Stops later RETURNs from moving the screen.
MOVE N	Moves the turtle N units forward.
TURN N	Causes the turtle to turn N degrees counterclockwise.
HOME	Returns the turtle, pointing to the right, to the center of the screen.

Screen Control

The first command ERASE causes the screen to be erased. Simply typing the word and then pressing the carriage return key <ret> causes the screen to be erased.

When a line is typed on the TV display screen and a carriage return key is pressed, the operator normally wishes the line just typed to move up — in the same manner that paper in a typewriter does; then any other characters that are entered will appear in the line immediately under the previous one. If there are several lines on the screen, all of them should move up when a carriage return key is pressed. This property of moving the entire screen up one line is called scrolling. Scrolling is convenient because it permits many lines to be read at one time on the screen. The pair of instructions SCROLL ON and SCROLL OFF control scrolling of the screen.

There are times when scrolling can present a problem — such as those times when instructions are given for figures to be drawn on the screen. Each instruction is executed when a

carriage return key is struck; but if that carriage return causes a scroll, the figure being drawn also scrolls upward. Then, after only a few instructions, the figure will be scrolled off the screen and lost. Consequently, scrolling is not wanted in most graphics applications.

The SCROLL ON and SCROLL OFF commands eliminate this problem by putting scrolling under user control. After the SCROLL OFF command is typed, subsequent carriage returns will not cause the screen to scroll. After the SCROLL ON command is typed, carriage returns will cause scrolling.

Turtle Commands

The remaining three commands in this group – MOVE, TURN AND HOME – always refer to an imaginary turtle that can move around on the screen, leaving a trail that can be seen as a line. This turtle will be used to draw figures and designs on the screen. The MOVE N command causes the turtle to move a certain distance and draw a line. The distance the turtle moves is determined by N, which can be any number.

On the average TV, the MOVE 5 command causes the turtle to move forward approximately one centimeter, leaving a line of that length behind. The relation between the number used for N and the distance moved on the TV depends on the TV; or, another way of saying this is that the units used for N in the MOVE instruction are arbitrary. To give an idea of the range of values that N might take, a MOVE 0.2 instruction causes the turtle to move the smallest perceptible amount on the TV, and a MOVE 128 causes the turtle to move from the far left edge to the far right edge of the TV screen, whatever its size.

The TURN N causes the turtle to turn N degrees. If N is a positive number, the turtle turns N degrees to the left (counterclockwise). If N is negative, the turtle turns to the right. Any reasonable number can be used for N: TURN 90 results in a 90° turn to the left; TURN 270 results in a turn of 270°, which is the same as TURN -90; TURN 360 causes the turtle to turn around completely and is the same as no turn at all.

When the TURN N command is typed and the carriage return key is pressed, there is no immediate change in the TV display. However, the next MOVE command will be at an angle relative to the last and determined by the intervening TURN command. For instance, Figure 1 shows the affect of a series of MOVE and TURN commands on the TV display. The triangle was drawn with the commands, MOVE 60, TURN 90, MOVE 45, TURN 126 and MOVE 75. The dotted lines, arrows and commands do not appear on the TV.

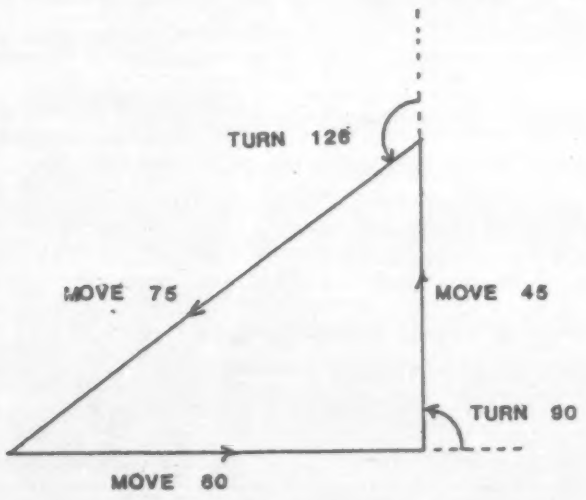


Figure 1. A Triangle Drawn with MOVE and TURN Commands.

The HOME command causes the turtle to go to its home, which is at the center of the screen facing to the right. HOME is used as a convenient way of having a standard place from which to start drawings.

These six commands (Table 1) can be used together to produce an unlimited number of line drawings. Figure 2 illustrates how a box could be drawn in perspective, using a series of 21 commands. At each stage, the commands listed on the left produce the drawings shown on the right.

```
(a)  SCROLL OFF
     ERASE
     HOME
     MOVE 30

(b)  TURN 60
     MOVE 25

(c)  TURN -150
     MOVE  30
     TURN -30
     MOVE  25

(d)  TURN -150
     MOVE  30
     MOVE -30

(e)  TURN  90
     MOVE  30
     TURN -90
     MOVE  30

(f)  TURN -30
     MOVE  25
     TURN -60
     MOVE  30
```

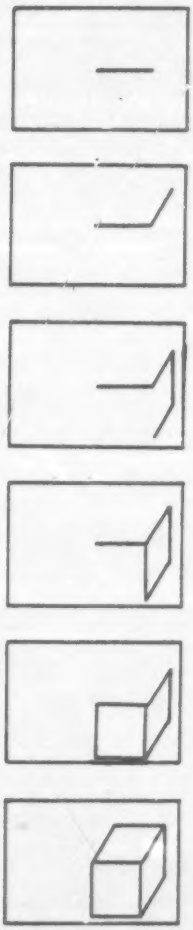


Figure 2. Drawing a Cube.

The preceding example illustrates that negative numbers can be used for N. A move of -30 results in a move backward of 30 units. The numbers do not have to be integers; the computer can understand commands like MOVE 2.0174. The result may not be drawn with the accuracy that such a precise number implies, because the smallest perceptible movement on the screen corresponds to MOVE 0.2.

EXAMPLE A: DRAWING A JET.

Given: The line drawing in Figure 3.

Figure 3.
Line Drawing
of a Jet.

Find: The commands that would draw this line drawing of a jet on the TV.

Solution: For simplicity, the length of the MOVE command is chosen to be equal to the length of the line of the figure (in centimeters). There may be some variation due to shrinkage of the illustration during printing, but if the bottom of the airplane is 11.2 cm long, then the turtle command is MOVE 11.2. To draw the next part of the figure – the sloping front – the turtle must be turned before another MOVE command is issued. The turtle would have to turn around to the left as shown. Since the turn is to the left, the angle is positive, and a protractor will show that

Example A. Continued.

it is numerically equal to 160°. This process of moving and turning is repeated until the entire figure is drawn. Note that the third turn (which is necessary before the rising part of the rudder is drawn) would be to the right. This is done with a TURN command with a negative value for the angle. The complete figure can be drawn by the following series of commands:

```
HOME
MOVE  11.2
TURN +160
MOVE   4.5
TURN +28
MOVE   4.9
TURN -52
MOVE   3.3
TURN  136
MOVE   3.5
```

ADDITIONAL GRAPHICS COMMANDS

The next series of commands is shown in Table 2. These additional graphics commands are necessary in many situations.

TABLE 2. ADDITIONAL GRAPHICS COMMANDS.

COMMAND	ACTION
DOT	Causes a dot to be drawn at the current turtle location.
PEN UP	Causes subsequent MOVE or JUMP instructions to leave no line on the screen. Cancelled by PEN DOWN.
PEN DOWN	Causes subsequent MOVE or JUMP instructions to leave a line on the screen. Cancelled by ERASER ON or PEN UP.
ERASER ON	Causes subsequent MOVE or JUMP instructions to erase the contents of any cell crossed. Cancelled by ERASER OFF and PEN DOWN.
ERASER OFF	Causes subsequent MOVE or JUMP instructions to neither erase nor leave a line. Cancelled by ERASER ON.
HOP N	Move forward N units without leaving a line. N can be any valid BASIC expression.
JUMP TO X,Y	Move the turtle from its current position to the coordinate position, X,Y, in screen units relative to HOME. X and Y can be any valid BASIC expression.

Pen Control

DOT causes a single dot to be drawn at the location of the turtle. For example, if the commands

```
ERASE
HOME
DOT
```

are entered in that order, the screen will be cleared – except

for a single dot in the center, which is the HOME position of the turtle.

PEN UP and PEN DOWN are a pair of instructions that control whether or not a line is drawn as a result of the MOVE and JUMP TO instructions. After the PEN UP command, MOVE commands do not result in a line being drawn. Once a PEN UP instruction has been issued, it can be cancelled by a PEN DOWN instruction. After PEN DOWN is issued, then all subsequent MOVE commands do leave a line.

EXAMPLE B: DRAWING WITH PEN CONTROL.

Given: The drawing in Figure 4.

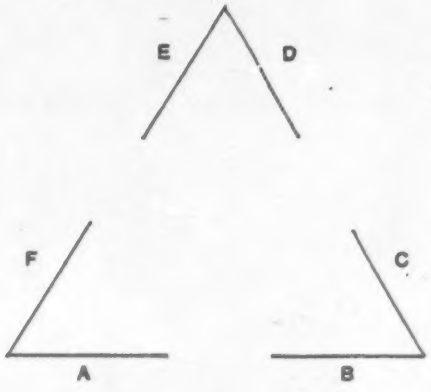


Figure 4.
Triangle Drawn with
Pen Control.

Find: The commands that would draw this triangular-shaped figure on the TV.

Solution: The following commands will complete the figure:

```
PEN DOWN
MOVE 3      } Draws line A

PEN UP
MOVE 2      } Hops to start of B
```

Example B. Continued.

PEN DOWN MOVE 3	Draws B
TURN 120	B-C corner
MOVE 3	Draws C
PEN UP MOVE 2	Hops to start of D
PEN DOWN MOVE 3 TURN 120 MOVE 3	Draws D and E vertex
PEN UP MOVE 2	Hops to F
PEN DOWN MOVE 3	Draws F

To draw a dashed line, the following series of commands could be given to the computer:

```
PEN DOWN
MOVE 2
PEN UP
MOVE 2
PEN DOWN
MOVE 2
PEN UP
MOVE 2 . . .
```

The ERASER ON and ERASER OFF pair of instructions control an imaginary eraser carried on the turtle. When the ERASER ON command is the last of the pair issued, then subsequent MOVE commands will erase any lines that cross the path of the turtle. To turn off this erasing feature, the ERASER OFF instruction can be issued.

The PEN and ERASER commands interact because it is illogical to have the pen down and the eraser on. Thus, after the ERASER ON command is issued, the pen is automatically up. Similarly, after a PEN DOWN command, the eraser is automatically off.

For example, to draw a line and then erase it, the following commands could be issued:

```
PEN DOWN
MOVE 20
ERASER ON
MOVE -20
```

The MOVE -20 command causes the turtle to move backward 20 units; and since the eraser is on when this happens, it erases the line previously drawn.

Other Turtle Moves

The HOP N command causes the turtle to move forward without leaving a line and without affecting whether the pen is up or down. Even if the pen is down, a HOP N command will cause the turtle to move forward N units and not draw a line. Thus, another way of drawing a dashed line would be to use the following commands:

```
PEN DOWN
HOP 2
MOVE 2
HOP 2
MOVE 2
HOP  2 . . .
```

For the student to understand the JUMP TO X,Y instruction, the concept of <u>screen coordinates</u> must be introduced. The TV

screen, as shown in Figure 5, can represent a piece of graph paper with the scales shown. The origin of the graph is in the center of the screen, and any point can be described by its X and Y coordinates (symbolized by X,Y). The X (horizontal) axis runs from -64 to +63.8. The Y (vertical axis runs from -51.2 to +51. On many TVs, not all of these coordinate points can be seen. The visible coordinates run from approximately -45 to +45 in the Y direction. As a result, the upper right-hand corner of the screen can be described by the coordinates 63.8, 45; and the lower left-hand corner can be described by the coordinates, -64, -45. As shown in Figure 5, part of the screen at the extreme top and bottom cannot be seen.

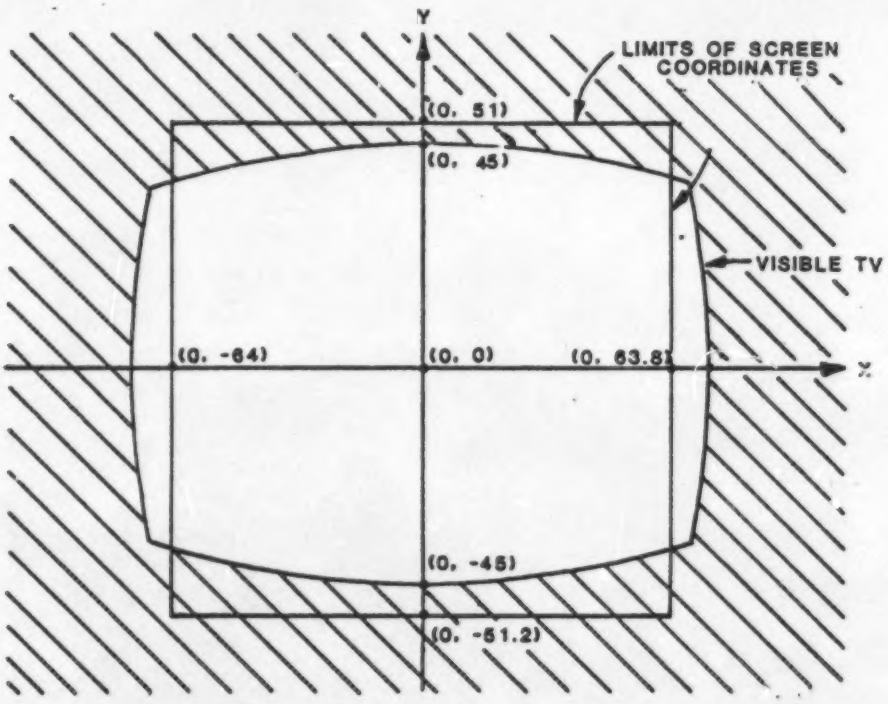


Figure 5. The Use of Coordinates in a TV Display.

The turtle can be instructed to jump to any screen coordinate by use of the JUMP TO X,Y command. The result of this command is affected by the PEN and ERASER commands in just the way that the MOVE instruction is. If the pen is down, then the JUMP TO command will cause a line to be drawn from the old position of the turtle to the coordinates given. If the eraser is on, the turtle will move from the old position to its new coordinates and erase any line along the way.

For example, the commands

```
HOME
PEN DOWN
JUMP TO 20,20
```

will cause a line to be drawn from the center of the screen to the coordinate position (20,20). Figure 6 shows the affect of a number of JUMP TO instructions.

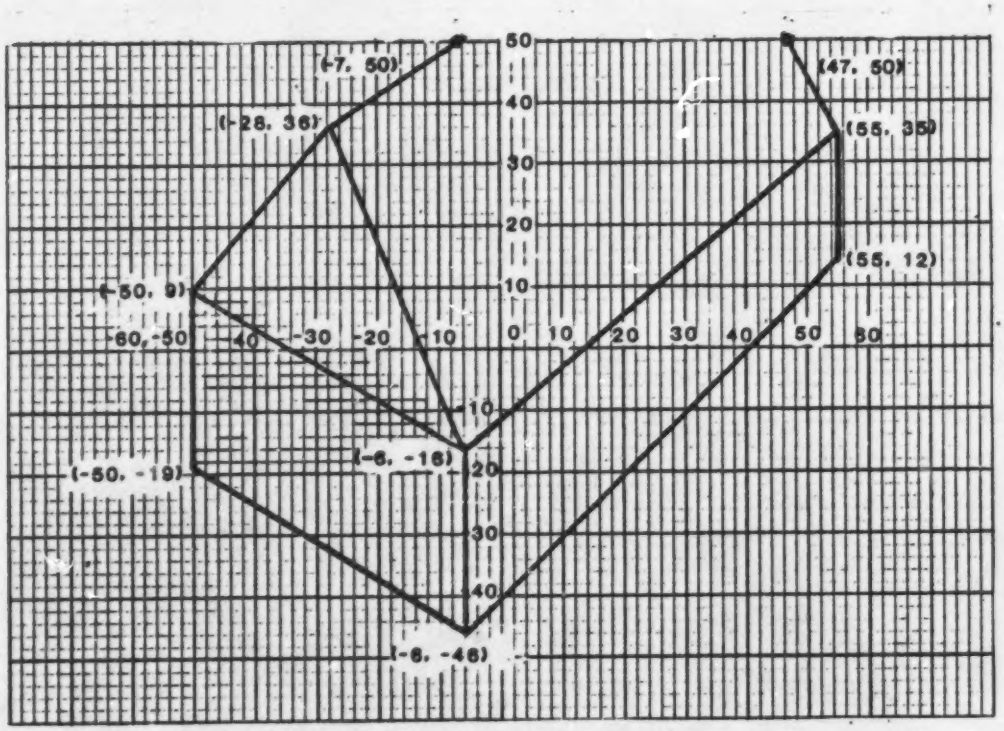


Figure 6. Result of Several JUMP TO Instructions.

The following instructions result in the drawing shown in Figure 6.

```
PEN UP
JUMP TO -50,-19
PEN DOWN
JUMP TO -6,-46
JUMP TO 55,12
JUMP TO 55,35
JUMP TO -6,-16
JUMP TO -28,36
JUMP TO -50,9
JUMP TO -50,-19
PEN UP
JUMP TO 55,35
PEN DOWN
JUMP TO 47,50
PEN UP
JUMP TO -7,50
PEN DOWN
JUMP TO -28,36
PEN UP
JUMP TO -6,-16
PEN DOWN
JUMP TO -6,-46
```

The 13 graphic commands discussed to this point are all that are necessary to produce striking affects. The examples above show that different combinations of these commands can produce interesting results.

COMPUTATIONAL COMMANDS

The commands previously covered did not include those that permit calculations; therefore, this section of the module outlines simple commands that allow the user to compute with numbers. Table 3 shows two commands and the basic arithmetic operations that are to be studied at this point.

TABLE 3. THE CORE COMPUTATIONAL COMMANDS AND OPERATIONS.

Commands	Action
LET variable = expression	Evaluates the expression and gives that value to the variable.
PRINT variable	Prints the value of variable.
expression 1 + expression 2	Adds the value of two expressions.
expression 1 - expression 2	Subtracts expression 2 from expression 1.
expression 1 * expression 2	Multiplies two expressions.
expression 1/expression 2	Divides expression 1 by expression 2.

THE LET INSTRUCTION

The LET instruction introduces the idea of a variable. A variable is a letter that has a numerical value. For instance, use of the instruction LET X = 7 gives a value of 7 to X. Once this command has been issued, then the letter X can be used in the same way numbers can be used. For instance, the pair of instructions

```
LET R = 15
MOVE R
```

results in assigning the number 15 to R, so that the MOVE command moves the turtle 15 units.

The word LET may be omitted from the LET command in the version of BASIC used in this module; therefore, LET R = 15 and R = 15 are equivalent instructions.

The use of the equal sign in the LET instruction is somewhat different from its usual use in algebraic expressions. The expression R = R + 1 is illogical mathematically because there is no value for R that will also be one greater than R. However, the expression is perfectly logical in BASIC, particularly when it is interpreted as an abbreviation for LET R = R + 1. This instruction takes the current value for R, adds 1 to it, and stores the result in R. For instance, if R were 15 before this instruction, it would be 16 afterwards.

In this BASIC, a variable can be any single letter or a letter followed by a single digit. These are valid variables: A, X1, C9, and Z. These are not valid: VAR, XI, C10, and $.

Arithmetic Expressions

The LET instruction can involve addition, subtraction, multiplication or division on the right-hand side of the equal sign. For instance, the following are four valid BASIC instructions, each of which assigns the number 12 to W:

```
LET W = 5 + 7
W = 3 * 4
W = 72/6
LET W = 15 - 3
```

The symbol for multiplication is a star (*), which is used instead of a period or an "x" to avoid possible confusion with other uses of those symbols. Division is indicated by a slash (/) because most typewriters used with computers generally do not have the standard division symbol.

More than one arithmetic operation can be combined in a single LET statement. For instance, the following is a valid BASIC instruction:

```
LET A = 4 + 6/2
```

When more than one arithmetic operation is combined in a single statement, there is sometimes some confusion about the meaning. In the example above, should the 4 and 6 be added to get 10 before dividing by 2 (in which case the result is 5), or should the 6 be divided by 2 first and then added to 4 to get 7? Which comes first, the division or the addition?

Parentheses

There are two ways to solve this problem. The better way to avoid questions about the order of arithmetic operations is to use parentheses to group sub-expressions together; then there can be no question:

```
LET A = (4 + 6)/2
```

The parentheses used above indicate quite clearly that the 4 and 6 must be added before they are divided by 2. Parentheses always indicate that the expression contained within them is to be evaluated and replaced by a single number.

Parentheses also can be used within parentheses to indicate the order of evaluation of an expression. For instance,

the expression

```
R = (4/(7 - 5) + 3) * 6
```

will be evaluated in the following steps:

```
R = (4/2 + 3) * 6
  = (2 + 3) * 6
  = 5 * 6
R = 30.
```

Parentheses can be used even when they are not absolutely necessary. For instance, in the expression

```
LET S = (3 + 7)
```

the parentheses are not needed at all. However, their use does not lead to an error. There are situations when a programmer may not know whether or not parentheses are required; but because no harm is done when they are used when not needed, the programmer would be well-advised to include them when in doubt.

Precedence Rules

A second way to resolve the question of which operation is performed first involves using the rules of precedence. In BASIC, all powers and roots and multiplications and divisions are performed before any additions or subtractions. Therefore, in a given instruction, all divisions are done first, then multiplications, then subtractions, and finally, additions. If this is as the programmer intends, then no parentheses are required.

EXAMPLE C: PRECEDENCE OF OPERATIONS.	
Given:	The expression L = 3/5 - 7 + 2 * 3.
Find:	The value for L.
Solution:	The computer will evaluate the expression in the following four steps, starting with division:

```
L = 0.6 - 7 + 2 * 3
  = 0.6 - 7 + 6
  = -6.4 + 6
L = -0.4
```

PRINTING

Once a lengthy computation has been performed, the programmer often wants to print out the result. This can be done with a PRINT command. For instance, the pair of commands

```
S = 7
PRINT S
```

results in 7 being displayed.

The value of more than one variable can be displayed with a single PRINT command as long as the variables are separated by commas. For example, the following three commands result in the numbers 6 and 10 being printed:

```
T = 6
R = T + 4
PRINT T, R
```

The PRINT command is so common that this BASIC interpreter gives it an abbreviation – the colon (:). As far as BASIC is concerned, the colon is equivalent to the PRINT command

when it appears as the first character in an instruction. As a result, the following two instructions are identical:

```
PRINT A, B
: A, B
```

Any expression can appear after a PRINT command; it does not have to be a simple variable. For instance, the command

```
: 18/0.5
```

results in printing the division of 18 by .5, namely 36. The LET and PRINT commands, together with the arithmetic operators, perform all the calculations of the hand calculator. In many respects, BASIC is more convenient than a calculator because complex equations can be entered in a single line with parentheses, which is similar to the way they appear in formulas.

EXAMPLE D: CALCULATOR MODE.	
Given:	$y = \frac{A + B}{3(C - D)}$ where: A = 3.7 B = 19.01 C = 8417 D = 7.31
Find:	Write a BASIC expression that would calculate and print the result.
Solution:	One way requires only one line: `PRINT (3.7 + 19.01)/(3*(8417 - 7.31))`

PROGRAMMING MODE

To this point, each command has been executed as soon as it was typed and the carriage return key pressed. This is called the immediate execution mode. Each command is executed immediately after it is entered.

The computer, however, can be used to store a series of commands without executing them. Then, when the programmer is ready, the entire series can be executed at once by typing the command RUN. This is called the programming mode.

LINE NUMBERS

Table 4 lists three commands related to the programming mode. In the programming mode, the instructions are almost the same as the immediate mode instructions decribed above; the only difference is that, in the programming mode, each instruction is preceded by a number called a line number. The computer executes the instruction with the lowest line number first and, unless told otherwise, continues to execute instructions in the order of the increasing line numbers.

For instance,

```
10 MOVE 30
```

has a line number of 10 and tells the computer that when it executes this line it is to move the turtle 30 units. When this line is first typed in, the computer takes no action, because the instruction is in programming mode. The computer simply stores this instruction along with any others that have been entered. When the command RUN is typed in, followed by carriage return, then the instructions with line numbers are executed.

TABLE 4. PROGRAMMING-RELATED COMMANDS.

Command	Action
SCR	Remove all stored commands.
RUN	Execute stored commands.
INPUT variable	Request a value for the variable from the user.

The command SCR is an abbreviation for SCRatch (or remove) any old program steps which may be left over from previous work. The series of commands

```
SCR
10 HOME
20 MOVE 20
RUN
```

causes the following action:

- The SCR removes any old commands from memory.
- Then the two numbered instructions are entered into memory, but not executed.
- Then when the command RUN is typed in, the computer causes the turtle to go to its HOME position and draw a line 20 units long.

If RUN were typed again, the turtle again would go home and draw a line

Notice that the line numbers do not have to start with 1, and not all the line numbers have to be used. The first instruction in this small program has line number 10, and the second has line number 20. It is good programming practice to leave many unused line numbers between instructions; then when errors are found, additional steps can be inserted.

Instructions may be inserted by giving the new steps line numbers between existing lines. For instance, if it were necessary in the program above to TURN 60 before the MOVE instruction,

this could be accomplished by typing this in the line later as follows:

```
15 TURN 60
```

Then when the program is run, the first line number 10 would be executed, then number 15, then number 20.

The advantage of a program is that the same procedure can be followed time and time again. When this is combined with the idea of variables, then the same procedure can be used with different values. The short program below draws a square with sides of length L:

```
10 MOVE L
20 TURN 90
30 MOVE L
40 TURN 90
50 MOVE L
60 TURN 90
70 MOVE L
80 TURN 90
```

As soon as the value for L is given and the program is run, a square will be drawn on the screen. For instance, to draw a square of size 5, one would type the following:

```
5 L = 5
RUN
```

To draw another square of size 10, one would type the following:

```
5 L = 10
RUN
```

In each case, line number 5 gave the value for L; then the program was run.

When a program is written, ability to see the instructions currently in the computer is helpful. This visualiza-

tion can be accomplished with the command LIST. When LIST is typed, the computer responds by listing all the program steps currently in memory. If the program is exceptionally long, it may be desirable to list just part of the program. For example, the command LIST 30 lists only those instructions after number 30, and the command LIST 70-90 lists only those instructions beginning with line 70 through line 90.

INPUT

A better way to get values into a program such as the previous example is to use the INPUT command. The command

```
5 INPUT "Give the size of the square"; L
```

requests the user to give the value for L, the length of a square. With this command in the program, when the program is run, the program types out,

```
Give the size of the square
```

and then waits until a number and a carriage return is typed.

There are three formats for an input command:

```
INPUT variable
INPUT variable, variable...., variable
INPUT prompt, variable,...., variable
```

The first waits for a single variable. The user is informed that the program needs a value for a variable because a question mark is typed out. In the second form, values for several variables are requested, each separated by commas or semicolons. Again, question marks are typed out to request those numbers.

The question marks can be confusing because they give no indication as to what values are needed; therefore it is clearer to type a message, called a prompt. The prompt is any message

enclosed in quotes. When the program is executed by a RUN command, the computer types the prompt ahead of the question mark.

EXAMPLE E: INTEREST RATES.

Given: The following program:

```
10 INPUT "Give the interest rate (in percent)"; I
20 INPUT "Give the amount invested"; P
30 P = P*(1 + 0.01*I)
40 PRINT "At the end of a year the investment is
          worth"; P
```

Find: Describe what happens when this program is run.

Solution: The first line requests the interest rate. When the user enters some number, its value is stored as I. The second line requests the principal and stores it as P. The third line calculates the value of the principal after one year. This value is equal to P*(1 + 0.01*I). The fourth line prints this answer with a message. One new item introduced to this program is the use of quotations within the PRINT command. As can be seen, this leads to a much clearer output. Each time it is executed, the program prints out the message, as well as the accumulated principal P.

SCIENTIFIC NOTATION

When the principal is large, the interest rate program (Example E) sometimes generates output that looks like the following:

0.31445E7

This is a floating-point number that uses scientific notation. BASIC uses floating point whenever it must deal with very large or very small numbers.

To interpret a floating-point number, find the E (which is an abbreviation for "Exponential"). The number before the E is called the mantissa. The number after the E is the exponent. The value of a floating-point number is the mantissa times 10 raised to the exponent power. Symbolically, it is as follows:

$$\text{mantissa E exponent} = \text{mantissa} \times 10^{\text{exponent}}$$

For example, the floating-point number above gives the following:

$$0.31745E7 = 0.31745 \times 10^{7} = 3,174,500$$

The exponent has the effect of moving (or floating) the decimal point to the right a number of digits equal to its value. If the exponent is negative, the decimal point moves left.

Other floating-points numbers are evaluated in the example that follows.

EXAMPLE F: FLOATING POINT NUMBERS.	
Given:	A = -0.7341E17
	B = 0.21438E-10
	C = -0.10074E-7
Find:	The decimal value of A, B and C.
Solution:	A = $-0.7341 \times 10^{17} = -73,410,000,000,000,000$
	B = $0.21438 \times 10^{-10} = 0.000000000021438$
	C = $-0.1007 \times 10^{-7} = -0.00000001007$

LOOPING

Instructions discussed so far are quite powerful. They permit shapes to be drawn on the screen and computations to be performed with lightning speed. However, to this point, every instruction given could be executed only once each time a program is run. If this were the only possible way to program, then programs that are complex would become quite long. Consider, for instance, the difficulty of drawing a circle. A circle can be drawn by a series of small MOVE and TURN commands. If the sequence MOVE 1, TURN 3 were repeated 120 times, the turtle would draw a figure that would look almost like a perfect circle; however, no one would wish to type in these two commands 120 times. BASIC provides two ways of repetitively executing one or more commands like these, as shown in Table 5.

TABLE 5. LOOPING COMMANDS.

Instruction	Action
GOTO N	The next instruction is at line number N. N can be a number or any valid BASIC expression.
FOR I = J TO K ⋮ NEXT I	The beginning of the range of a FOR, NEXT loop. The index I can be any variable. J and K can be any numbers or valid BASIC expressions. Marks the end of the range of the FOR, NEXT loop with the index I.

THE GOTO COMMAND

The simplest way to loop is to use the GOTO N command. When a GOTO N command is executed, the next instruction executed is the one with line number N. For instance, a circle could be drawn with the following three instructions:

```
10 MOVE 1
20 TURN 3
30 GOTO 10
```

Then, when the RUN command is given, the computer would repeat endlessly the TURN and MOVE commands; and as a result, a circle would be drawn after 120 repetitions. There is nothing in this program to stop the computer after a complete circle; so it will continue redrawing the circle.

There is a problem with the preceding example, in that the computer gets "hung up" endlessly going around the 3-line loop. The only way to stop the computer when it is trapped in a loop of this kind is to press the CONTROL key and the C key simultaneously to enter a special command called CONTROL C; this stops or "breaks" the execution of the program.

LOOPS

To avoid getting trapped inside the loop, it would be better to execute the two commands exactly 120 times; this can be done with the FOR and NEXT commands. FOR and NEXT are two instructions that must be used together. (If one is used, the other must be used.) When they are separated in a program, all the instructions between can be repeated. The part of the program that might be executed between the FOR and NEXT instructions is called the range.

The number of times instructions in the range of a FOR, NEXT pair are repeated depends on more information in the FOR instruction. This is illustrated in the following program:

```
10 FOR I = 1 to 120
20 MOVE 1
30 TURN 3
40 NEXT I
```

In this program, the instructions for lines 20 and 30 are executed 120 times. Each time they are executed is called an iteration. The first time through the loop the variable I is set to 1; the second time, 2; and so forth. Each time through the loop the variable is increased by 1, and the process is continued until I takes on the last value – in this case 120. After the last time through the loop with I = 120, any instruction following the loop would be executed. In this particular case there are none, and the program stops.

The NEXT instruction marks the end of the range. The variable I could have been any other variable, but the same variable must be in both the FOR and NEXT instructions. This variable is called an index.

The variable following the word "FOR" is the index. The index is first given the value of the expression to the right of the equal sign. Then, each time through the loop, the index is increased by one. This process continues until the value of the index exceeds the value of the expression after the word "TO". For instance, the loop that starts

```
FOR K = 7 TO 3 * 6
```

would start with K = 7 and repeat until K = 18. The loop with K = 18 would be executed, but not one with K = 19.

In the example above, the index I is not used anywhere in the program except to count the number of loops. However,

this variable is available to the program and could be used to great advantage in many situations. The program in Table 6, for example, calculates the value of an investment made today after a given number of years at a given rate of return. The index is N and is used in the printout.

TABLE 6. THE INVEST PROGRAM.

```
10 INPUT "Give the interest rate (in percent):"; I
20 INPUT "Give the amount invested"; P
30 INPUT "For how many years do you want to invest?"; Y
40 FOR N = 1 to Y
50 P = P*(1 + .01*I)
60 PRINT "After"; N; "years, the principle is worth"; P
60 NEXT N
```

In line 30, the program asks for the number of years the investment is to be made and stores the value given in Y. Because Y is in the FOR instruction, the program repeats the simple interest calculation that number of times. It does this by looping through the calculation once for each year that the money will be invested. When the program is run, the results can be seen as Figure 7.

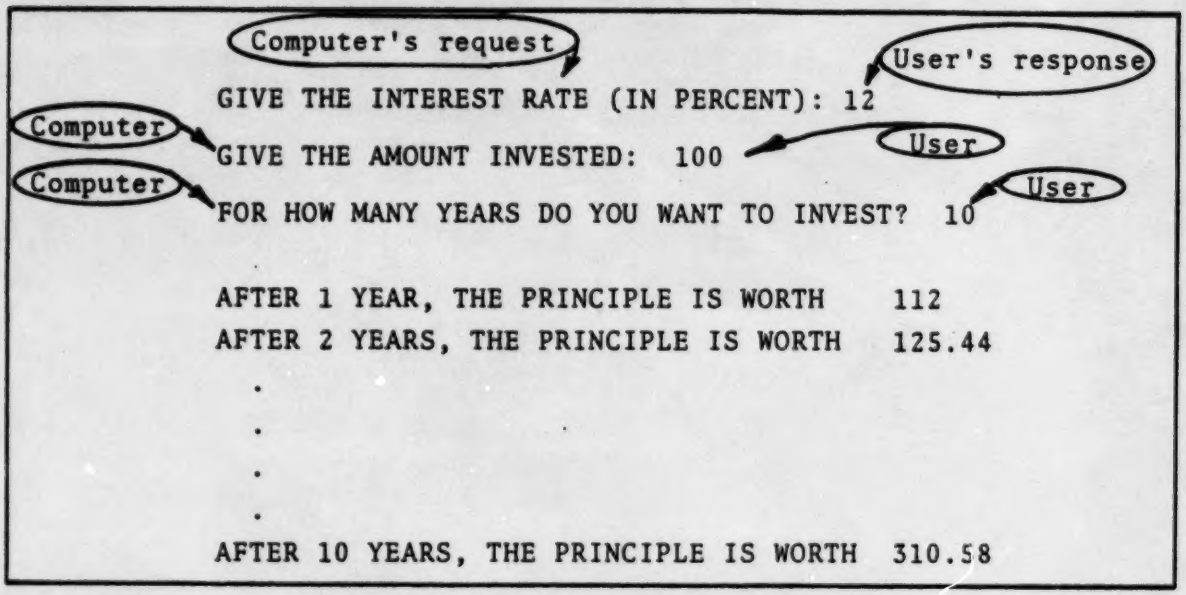

Computer's request
User's response
GIVE THE INTEREST RATE (IN PERCENT): 12
Computer
GIVE THE AMOUNT INVESTED: 100
User
Computer
FOR HOW MANY YEARS DO YOU WANT TO INVEST? 10
User
AFTER 1 YEAR, THE PRINCIPLE IS WORTH 112
AFTER 2 YEARS, THE PRINCIPLE IS WORTH 125.44
AFTER 10 YEARS, THE PRINCIPLE IS WORTH 310.58

Figure 7. One Run of the Investment Program, INVEST.

Combined with the previous instructions, the FOR, NEXT pair can generate some extremely sophisticated programs. One powerful idea is to place one FOR, NEXT pair inside the range of another pair. This is called nesting. Nesting is possible as long as the indices for the two FOR, NEXT pairs are different.

EXAMPLE G: NESTED LOOPS.

Given: The following program:

```
10 HOME
20 PEN DOWN
30 FOR I = 1 TO 10
40 HOP 15
50 FOR J = 1 TO 4
```

Example G. Continued.

```
            60 MOVE 10
            70 TURN 90
            80 NEXT J
            90 NEXT I
           100 END
```

Find: The shapes drawn on the screen.

Solution: The inner loop consists of lines 50, 60, 70 and 80. In each of these loops, one line is drawn and the turtle is turned 90°. Repeating this four times draws a square. The outer loop extends from lines 30 to 90. Each time through this loop, the turtle hops forward 15 and draws a square (by executing the inner loop). By repeating this ten times, 10 squares are drawn in a row.

Note the END instruction at line 100. A good practice is to include this statement at the end of each program, as it definitely tells the computer to stop at the end of the program. If there were a higher line number with instructions still uncleared from a previous program, the computer would proceed and execute that command were it not for END.

DISK CONTROL

Once a program has been developed and run properly, it is useful to be able to store it for later use. This requires the disk control commands listed in Table 7.

TABLE 7. DISK CONTROL COMMANDS.

Instruction	Action
NAME	Computer responds with the current program name.
NAME P	Computer gives the name P to the current program. P can be any valid program name.
SAVE	Save the current program, using the current program name.
OLD	Requests a program to be loaded in from the disk.
DIR *.BSC	Lists all BASIC programs on the disk.
ERA P.BSC	Erases the BASIC program P. The name P can be any valid program name.

When a program is used at a later date, it is referred to by name; therefore, before the program is saved, it must be given a name. Anytime BASIC is begun, it asks the question,

NEW OR OLD?

An old program is one that is already stored in disk; whereas a new program is one that will be generated, using BASIC. In either case, the name of the program is requested. When the name of an old program is given, the program will be brought in from memory. If it is a new program, it is up to the programmer to invent a name for that program. The name of a program must start with some letter and be not more than seven characters long and have no spaces. MYPRG (my program) is a valid name for a program, but 3PRG is not, nor is MY PRG.

To find out the name of the current program, the command NAME can be typed in at any time. The computer responds by typing the current name of the program. To change the name of the program, NAME is typed, followed by the new name of the program. Thus, the typing of

```
NAME JIM
```

causes the current name for the program to be JIM.

To save a program, simply type the command SAVE. This causes the computer to save all the instructions with line numbers on a disk using the current program name. If the disk already has a program with that name, the old program is lost and the new one written over it. To retrieve a program from disk, the command OLD can be typed at any time. The computer will respond by requesting the name of the program. As soon as this is given, the disk will be searched for a program with that name. If such a program exists on the current disk, it will be loaded in and made available. Any program previously in memory will be lost.

Sometimes it is useful to see what programs are on the disk. This can be done with the following command:

```
DIR *.BSC
```

If there are programs that should be erased from disk, the command

```
ERA P.BSC
```

can be used. As written, this erases the program name P. Any valid program name can be used. The ".BSC" in the two commands above informs the computer that only BASIC programs are of interest. This is actually part of the name of any

BASIC program and is supplied automatically by BASIC. This can be seen when the command NAME is used, because the ".BSC" is returned with the program name.

EXAMPLE H: SAVING AND USING PROGRAMS ON DISK.	
Given:	The program STAR which resides on disk.
Find:	The steps necessary to add the line 30 TURN 144 to the program; then rename the program STAR5 and save it on the disk.
Solution:	The command OLD STAR gets the program stored in the computer. Then the new line can be added by typing the following: 30 TURN 144 The program can be renamed using the following command: NAME STAR5 This program can be saved under its new name by typing the command SAVE

EXERCISES

1. Describe the figure drawn by the following programs.

a.
```
10 HOME
20 MOVE 10
30 TURN 120
40 MOVE 10
```

b.
```
10 HOME
20 TURN 120
30 MOVE 20
40 GOTO 20
```

c.
```
10 HOME
20 MOVE 2
30 PEN UP
40 MOVE 2
50 DOT
60 MOVE 2
70 PEN DOWN
80 GOTO 2
```

d.
```
10 FOR I = 1 TO 20
20 MOVE I
30 TURN 60
40 NEXT I
```

e.
```
10 PEN UP
20 FOR X = 50 TO 50
30 FOR Y = 50 TO 50
40 JUMP TO X,Y
50 DOT
60 NEXT Y
70 NEXT X
```

2. Describe the results of running the following programs:

   a.
```
10 PRINT "PLEASE ENTER A NUMBER"
20 INPUT A
30 PRINT A*(A + 1)
40 GOTO 10
```

   b.
```
10 A = 3E7
20 B = 0.1E-2
30 D = A/B*4-3
40 PRINT A,B,D
```

3. Write a program that does the following:
   a. Draws a triangle with 3 equal sides.
   b. Draws two touching circles.
   c. Requests two numbers and prints their product.
   d. Requests a number and draws a grid that fills the screen with that many horizontal and vertical lines.

LABORATORY MATERIALS

A disk-based microcomputer with CP/M and BASIC.
1 full-size floppy disk.
1 Metric ruler.
1 Protractor.

LABORATORY PROCEDURES

LABORATORY 1: GRAPHICS.

1. Read BASIC into the computer. Apply power to the computer. Place the disk in the disk reader. Press the RESET button. Type BASIC <ret> to read BASIC from the disk. Name the program as requested.

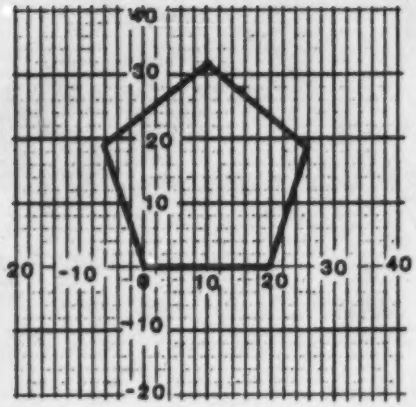


Figure 8. A Pentagon Used in Laboratory 1.

2. Recreate the pentagon depicted in Figure 8 on the TV, using MOVE and TURN instructions. Several ways have been outlined that could be used to create a line drawing using the TV; the easiest is to type a series of MOVE and TURN commands in immediate mode. Measure the angles and distance in Figure 8. Translate these into immediate mode MOVE and TURN commands. Enter the commands in Data Table 1, under Step 2. Enter these commands on the computer. Make any corrections necessary, and record the successful commands.
3. Create the pentagon in Figure 8, using JUMP TO commands. Figure 8 has been drawn on a grid to simplify finding the coordinates of the ends of the straight lines. Determine these coordinates, and use them to draw the figure with JUMP TO commands. Record in Data Table 1. Enter these commands on the computer. Make any corrections necessary, and record the successful instructions.

4. Create the pentagon in Figure 8, using looping. The five angles and five sides in Figure 8 are all the same; therefore, instead of five identical MOVE and five identical TURN commands, only one of each is needed in a loop that is repeated five times. Write a program that accomplishes this, using the GOTO command, and record in Data Table 1, Step 4. Enter and run program. Use CTL-C (press CONTROL and C keys together) to stop the program. Record the successful program. Modify the program, using the FOR, NEXT pair so that just the five sides are drawn. Enter, run and record this program. NOTE: If a mistake is made entering an instruction, any of the following three steps can be taken to correct the error:

   a. Cancel (erase) the entire line by pressing CTL-U (CONTROL and U keys).
   b. Delete the last character by pressing the DELETE key.
   c. Retype the line.

5. Enter a polygon drawing program. The program below asks for the number of sides desired and then draws a polygon with that many sides:

```
 10 ERASE
 20 SCROLL OFF
 30 INPUT "GIVE THE NUMBER OF SIDES"; N
 40 PEN UP
 50 HOME
 60 JUMP TO 0, -50
 70 PEN DOWN
 80 MOVE 150/N
 90 FOR I = 1 TO N
100 TURN 360/N
110 MOVE 300/N
120 NEXT I
```

Enter this program, name it POLY, save it and run it for the following number of sides: 3, 5, 17, 100. Describe the results in Data Table 1, Step 5.

6. Improve POLY. POLY is inefficient as written because two unnecessary divisions (in Steps 100 and 110) are required each time through the loop. These divisions should be done once before the FOR instruction and the results stored in new variables. Make these changes and rename the program POLY1 and save it. Time both POLY and POLY1, drawing a 100-sided polygon. How much time was saved by eliminating 200 divisions? Determine from this how long one division takes. Record the estimate and the changes in the program.

7. Modify FIELD. Enter, save and execute the following program called FIELD:

```
 10 HOME
 20 ERASE
 30 FOR X = -6 TO 5
 40 FOR Y = -4 TO 4
 50 PEN UP
 60 JUMP TO 10*X, 10*Y
 70 PEN DOWN
 80 FOR I = 1 TO 5
 90 MOVE 7
100 TURN 144
110 NEXT I
120 NEXT Y
130 NEXT X
```

Describe the TV display that FIELD generates. Modify FIELD to draw squares instead of stars. Rename this program SQF and save it. Record the changes required. Modify FIELD to draw the jet in Figure 3 instead of stars. Rename this program JETF and save it.

LABORATORY 2: COMPUTATIONS.

1. Use the calculator mode to evaluate expressions. Load BASIC into the computer and use it to evaluate algebraic expressions in its calculator mode. Use A = 0.3, B = 0.000017, C = 340,700,000, and D = 0.31415. Calculate and record the results in Data Table 2.
   a. $74 \cdot 35 \cdot 64$
   b. $A\ (3B + \frac{20D}{C})$
   c. $\frac{5C}{B + \frac{A}{30D}}$
   d. $\frac{ABC}{D}$
2. Enter INVEST. Enter and execute the investment program in Table 6. Name it INVEST and save it under that name. Use it to calculate the term value of $1000 invested at 5% after 25 years. Use the program to decide whether twice the interest rate doubles the interest earned. Record the results in Data Table 2, Step 2.
3. Modify INVEST. INVEST prints out the principle value after each year. Suppose all that information is not needed; only the final value. Modify the program to omit all the intermediate printing. Rename the program INVEST1 and save it. Record the changes required.
4. Calculate the heat flow through a wall. The amount of heat lost through a wall depends on the following:
   a. The thickness, T, of the wall in meters.
   b. The height, H, and length, L, of the wall in meters.
   c. The temperature difference, D, between the inside and outside of the wall in degrees Celcius.
   d. How well the wall conducts, measured as its specific conductivity C.

   The value of Q, the heat lost in watts, is as follows:

$$Q = \frac{CLD}{T}$$

Typical values for the specific conductivity are given in Table 8.

TABLE 8. SPECIFIC CONDUCTIVITIES.

MATERIAL	SPECIFIC CONDUCTIVITY C (watts/meter^{-1}/degree^{-1})
WOOD (Pine)	0.11
GLASS	0.7 – 0.9
CONCRETE	1.3 – 3.6
GLASS WOOL INSULATION	0.04
STONE (Granite)	1.9 – 4.0

Write and save a program, called COND, that requests input values for each of the five variables and then prints the resulting heat loss. Use COND to find the heat lost through a wood wall 3m high, 10m long, and 0.2m thick when it is 25°C inside and 0°C outside. Repeat the calculation for concrete and glass wool insulation. Record the results.

DATA TABLES

DATA TABLE 1: GRAPHICS.

STEP 2: MOVE AND TURN INSTRUCTIONS

Immediate mode commands used to recreate Figure 8:

_____	_____
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

STEP 3: JUMP TO COMMANDS

Immediate mode JUMP TO commands used to recreate Figure 8:

_____
_____
_____
_____
_____
_____
_____

STEP 4: LOOPING

Program using GOTO that recreates Figure 8:

_____
_____
_____
_____
_____

Data Table 1. Continued.

STEP 5: PROGRAM USING A FOR, NEXT PAIR TO RECREATE FIGURE 8:

Describe the results: _____

STEP 6: IMPROVE POLY

Estimated time per division: _____

Changed lines in POLY: _____ _____

_____ _____

STEP 7: FIELD

Describe the display FIELD generates: _____

_____

Changes required to generate squares:

_____

_____

_____

_____

Changes required to draw jets:

_____ _____

_____ _____

_____ _____

_____ _____

DATA TABLE 2: COMPUTATIONS.

STEP 1: CALCULATOR MODE

Results of calculations:

a. _____

b. _____

c. _____

d. _____

STEP 2: INVEST

Values of $1,000 after 25 years at 5%: _____

Amount gained: _____

Value of $1,000 after 25 years at 10%: _____

Amount gained: _____

STEP 3: INVEST MODIFICATIONS

Changes required to print only the final principle:

_____

_____

_____

STEP 4: HEAT FLOW

Record the program, COND:

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

_____

Data Table 2. Continued.

Heat loss through –

a. Wood: _____ watts

b. Glass: _____ watts

c. Concrete: _____ watts

d. Glass wool insulation: _____ watts

e. Stone: _____ watts

REFERENCES

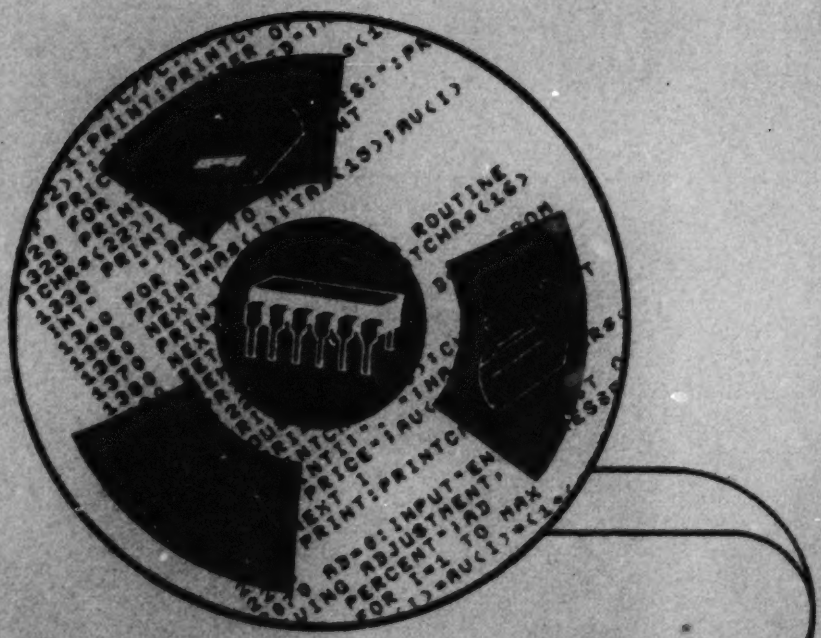
Albrecht, Bob; Finkel, Leroy; and Brown, Jerald R. BASIC for Home Computers. New York: John Wiley & Sons, Inc., 1978.

BASIC User's Manual. TERC.

Dwyer and Critchfield. BASIC and the Personal Computer. Reading, MA: Addison-Wesley Publishing Company, 1979.

ENERGY TECHNOLOGY

CONSERVATION AND USE



MICROCOMPUTER OPERATIONS

MODULE MO-07

BASIC PROGRAMMING

CENTER FOR OCCUPATIONAL RESEARCH AND DEVELOPMENT

INTRODUCTION

This module introduces a number of programming concepts that make BASIC a powerful and useful language. The previous module in this series is an introduction to BASIC programming which describes a core set of BASIC instructions. Many problems can be solved with programs using just those instructions; however, this module will show that there are problems that are either very difficult or impossible to program without the aid of some additional instructions and programming techniques.

This module introduces conditional statements, functions, subroutines, indexing, and arrays; and it shows how these ideas can be applied to specific calculations.

PREREQUISITES

The student should have completed Modules MO-01 through MO-06 of Microcomputer Operations.

OBJECTIVES

Upon completion of this module, the student should be able to:

1. Define the action of the following commands:
   a. DIM
   b. IF
   c. GOSUB
   d. RETURN

e. DATA
f. REM
g. STOP
h. APPEND
i. READ
j. DATA

2. Define an indexed variable; describe how to access and alter its contents.
3. Define functions and subroutines; and describe the action of the following functions:
   a. ABS
   b. INT
   c. RND
   d. SGN
   e. SIN
   f. TAN
   g. COS
   h. SQR
4. Determine the action caused by programs that use the commands listed above, indexed variables and the functions listed above.
5. Write short programs using these commands and data structures.
6. Analyze the computer resources required for computation and graphing.

CONDITIONAL STATEMENTS

There are many situations in programming where one part of the program can usually be executed, but under certain conditions another program step is needed. This is called branching; that is, the program must go one of two possible ways, depending on the result of some calculation or input. The IF...THEN statement is an instruction in BASIC that permits branching. A typical use is shown below:

```
300 IF N = 7 THEN Y = 8
```

This statement gives Y a new value - namely 8 - only in the case where N = 7. If N does not equal 7, Y is unchanged. Another example follows:

```
10 IF R > 3 THEN GOTO 400
```

In this case, if R is greater than 3 (e.g., 700 or 3.001), then the next instruction to be executed will be at line number 400. On the other hand, the GOTO statement will be ignored if R is equal to or less than 3 (e.g., 0, -6,000 or 3). In this case, the next instruction will be the one following the IF...THEN statement.

PARTS OF THE IF INSTRUCTION

There are four parts to the IF...THEN instruction.

These are illustrated in the complete IF statement below:

```
1        2          3          4
IF  conditional  THEN  branch instruction
```

The first part is the IF word itself; this is absolutely required. The second part is called a conditional statement, which, in most simple cases, will involve comparing two expressions with some relational operation. In the latter example given, R and 3 were compared with a relational operation >. The result of a conditional expression is always either true or false. In the example, either R is greater than 3 or it is not; in the first example, either N was equal to 7 or it is not. If the condition were true, then the branch instruction is executed.

The seven kinds of relational operations allowed in BASIC conditional expressions are listed in Table 1

TABLE 1. RELATIONAL OPERATIONS.

Relation	Meaning
>	Greater than.
<	Less than.
=	Equal to
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to.

The third part of the IF statement is the word THEN. The primary purpose in typing the word THEN is to mark the end of the conditional statement, as well as to make the instruction

readable.

After THEN comes the fourth part of the IF statement, the branch instruction. The branch instruction can be any valid BASIC instruction. It is only executed if the result of the conditional expression is true. If the result of the conditional expression is false, then the branch instruction is ignored and the computer executes the next instruction.

There are situations in which the calculations needed if the conditional is true can be done in a single instruction. In this case, the single instruction can just be the branch instruction within the IF statement. There are many situations, however, in which extensive calculations must be performed if the conditional is true. In this case, a GOTO (Go to) statement can be used for the branch instruction.

The GOTO statement starts the computer executing a totally different part of the program, which may contain a large number of steps. This situation is so often encountered that an abbreviation has been developed for BASIC. As branch instruction, GOTO 300 and 300 are the same. This is, the word GOTO may be omitted. If it is omitted, then all that is required is the line number to jump to if the condition holds.

An example of the use of the implied GOTO statement is the following instruction:

```
600 IF (X + 7) > Y * Z THEN 200
```

If the condition holds (i.e., if X + 7 is indeed greater than the product of Y times Z), then the next instruction to be executed will be at line 200. On the other hand, if the condition fails (i.e., if X + 7 is not greater than the product of Y and Z), then the next instruction will be the one immediately following line 600.

Below is an example of how IF instructions can be used in a complete program:

```
10 INPUT "GUESS MY NUMBER"; G
20 IF G = 31 THEN 60
30 IF G < 31 THEN PRINT "TOO LOW"
40 IF G > 31 THEN PRINT "TOO HIGH"
50 GOTO 10
60 PRINT "CORRECT!"
70 END
```

Line 10 requests the user to guess a number the computer is "thinking of." The guess is stored in the variable G. If the guess is 31, it is correct; and in this case, line 20 uses an implied GOTO to transfer control to line 60. Line 60 prints CORRECT!; then the program stops. If the guess was not 31, line 30 checks to find out if the guess was less than 31. If so, TOO LOW is printed. Similarly, line 40 checks to find out if the guess was greater than 31. In this case, TOO HIGH is printed. In either case, line 50 transfers control to line 10, which requests another guess.

EXAMPLE A: CONDITIONAL EXPRESSIONS.
Given: The following program: 10 I = 1 20 PRINT I, 3 * I - 7 30 I = I + 1 40 IF I < 5 THEN 20
Find: The output generated by this program when it is run.
Solution: This program illustrates a way of using conditions

Example A. Continued.

to cause looping as the FOR, NEXT pair can. I is set to 1 in line 10, so line 20 prints the following:

```
 1 -4
```

Then I is increased to 2 in line 30. Since this is less than 5, line 20 is again executed. This now prints:

```
 2 -1
```

The loop is executed again, printing:

```
 3  2
```

and again, giving:

```
 4  5
```

After printing this, I is increased to 5 in line 30. Then line 40 discovers I is no longer less than 5, so the next instruction is executed. There are no other instructions, so the program stops.

INDEXING

Consider the problem of storing a large amount of data from a single source. These data might represent the temperature at a particular site over several weeks; there might be thousands of readings which need to be entered into the computer and analyzed.

There is no convenient way - using the BASIC commands discussed to this point - to handle all these data. There are two things wrong with assigning each reading to a different variable: 1) there might not be enough variables, and 2) a separate instruction would have to be written for each different variable, which could lead to unnecessarily long code.

The way to solve these problems is to use an indexed variable; this is, a variable that can store many different numbers. These numbers are stored in order and can be accessed by using an index that indicates which number is desired.

For instance, "A(3)" means the third number stored in the variable A, and "R(375)" means the 375th number stored as variable R. Using indexed variables, 10,000 temperature-data points could be stored by using one indexed variable. An indexed variable can be used anywhere a number or a regular variable can be used. It must always be followed by its index; which consists of parentheses around a number or expression. Thus, T(9) = 4 is a LET statement that assigns 4 to the 9th number in the T variable, and B = T(9) *3 multiplies the 9th T number by 3 and stores the result in S. The statement

```
IF T(9) > B(1) THEN 30
```

goes to instruction 30 if the 9th T is larger than the first B.

The following states the general form for an indexed variable:

```
variable    (expression)
```

That is, any valid BASIC variable may be used as a simple variable or as an indexed variable, but its use must be consistent within a given program. It is recognized as indexed by the parentheses that immediately follow the variable. Any valid expression that can be evaluated can be within the parentheses: a number, a variable, or an expression. Thus, Z9(4), R(A), T(3*I-1), and R(T(3)-1) can all be vaild indexed variables. In the last example, the index of R is one less than the value of the third number in T.

DIMENSIONS

The BASIC language must save addresses in which to store an indexed variable; if 300 readings are going to be stored in T, BASIC must make room for 300 numbers in memory.

This allocation of memory is done with a DIM statement. DIM is an abbreviation for "DIMension." The statement

```
10 DIM T(300)
```

reserves 300 locations for the indexed variable T. The 300 in this example is the dimension of T. Every indexed variable must be dimensioned, using a DIM statement, before it is used in a program.

Once a variable appears in a DIM statement, it must always be indexed. For instance, A and A(3) cannot be used in a program: A is a simple variable and A(3) is indexed; the same variable cannot be both.

The index of a variable must be between 0 and the dimension stated in a DIM statement. For instance, the following are invalid for T as dimensioned above: T(-1), T(1000), and T(-30). The program in Table 2 illustrates the use of an indexed variable.

TABLE 2. USE OF INDEXED DATA.

```
 10   DIM D(1000)
 20   REM ENTER DATA
 30   I = 0
 40   I = I+1
 50   PRINT "ENTER DATA POINT", I
 60   INPUT D(I)
 70   REM CHECK FOR MORE DATA
 80   INPUT "TYPE Ø TO ENTER ANOTHER POINT"; S
 90   IF S = 0 THEN 40
100   REM THE CURRENT I IS THE NUMBER OF DATA POINTS
110   N = I
120   REM SEARCH FOR THE MAXIMUM
130   M = -1E64
140   FOR I = 1 TO N
150   IF D(I) > M THEN M = D(I)
160   NEXT I
170   REM NORMALIZE DATA BY DIVIDING BY THE MAXIMUM
180   FOR I = 1 TO N
190   D(I) = D(I)/M
200   NEXT I
```

The command REM is introduced in this program. REM is an abbreviation "REMark," and is used to insert comments into the program to make it readable. REM commands are ignored by BASIC. It is good practice to use REM comments with longer programs to clarify what each section of the program accomplishes; but, of course, the program will run without any REM statements.

This program asks the user to enter up to 1000 data points

that are stored in D. Those points are searched for the largest data point, then all are divided by this largest value. This process, called normalizing, results in data that is proportional to the original data, but all points are less than or equal to 1. The normalized data are also stored in D. This program is not very useful as written - it is a logical beginning to a longer program that might perform additional computations on the data or graph the results.

FUNCTIONS

A function is a rule that relates two variables. One simple rule is "take the positive square root." This defines the square root function. If this rule is applied to 9, the result is 3; thus, 9 and 3 are related through the square root function.

FORMAT OF FUNCTIONS

BASIC has certain built-in functions. For instance, the square root function is called SQR in BASIC. To print the value of the square root of 9, type the following command:

```
PRINT SQR(9)
```

The computer will respond by printing 3. The general form of a function in BASIC is as follows:

```
functionname  (expression)
```

The functionname can be any of those listed in Table 3. The

expression is called the argument of the function; it can be any valid BASIC expression that can be evaluated to give a number.

When BASIC encounters a function, it evaluates the function and replaces it with a number. For instance, SQR(9) becomes 3. This number is called the value of the function. This value can be used anywhere that a number or expression is allowed in BASIC. Some examples of how functions can be used are shown below:

- In a simple LET statement, to set Y to 3:

```
Y = SQR(9)
```

- In a compound LET statement:

```
Y = 6/SQR(9)+1
```

- As an index:

```
PRINT R(SQR(9))
```

(This assumes R has been dimensioned and that R(3) is defined.)

- With a variable argument:

```
Y = SQR(X)
```

(If the statement X = 16 appeared earlier, then Y would be given the value 4.)

- With a function argument:

```
Z = SQR(SQR(16))
```

TABLE 3. BUILT-IN BASIC FUNCTIONS.

INT	Takes the integer part of the argument.
RND	Generates a random number.
SGN	Finds the sign of the argument. This function is as follows: -1 if the argument is negative

Table 3. Continued.

	0 if the argument is zero +1 if the argument is positive
ABS	Takes the absolute value of the argument.
SQR	Takes the square root of the argument.
SIN, COS, TAN	Trigonometric functions. All assume the argument is in radians (360° corresponds to 2π or 6.28 radians).

OTHER FUNCTIONS

The INT function gives the nearest integer below its argument. For instance, INT(3.17) is 3; the fraction part, .17, is dropped. If the argument is already an integer, there is no change. Thus, INT(37) gives 37.

For negative numbers, INT may seem a little peculiar, since INT(-3.1) gives -4; but this fits the rule, since -4 is the next integer below -3.1.

The RND function generates a random number between 0 and 1. A computer is actually too predictable to generate a totally unpredictable number. However, the numbers generated by RND are random in a statistical sense and, under most conditions, impossible to predict. Because they are almost random, but still reproducible, they are sometimes called pseudorandom.

The RND function requires an argument so BASIC can recognize it as a function. However, the value of the argument does not matter.

INT and RND can be used together to generate random

integers within a specified range. Consider the following statement:

```
Y = 1 + INT(6*RND(1))
```

RND is given the argument 1 for simplicity. (Some BASIC interpreters require RND(Ø) to be entered.) It returns any value between 0 and 1. Multiplying this by six gives a random number between 0 and 6. This forms the argument for the INT function, which returns only the integer part of the random number. The result is the same as throwing a dice; one of the integers 1, 2, 3, 4, 5, or 6 will be generated at random.

The Guess My Number program can be vastly improved by generating a random number to guess instead of always using 31. The improved program shown in Table 4 generates some number R between 1 and 1000.

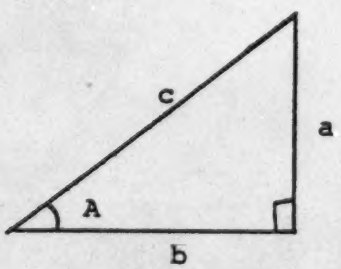
TABLE 4. GUESS A NUMBER GUESSING GAME.

```
10   R = HINT(1000*RND(1))
20:  "I AM THINKING OF A NUMBER BETWEEN 1 AND 1000"
30   INPUT "GUESS THE NUMBER"; G
40   IF G=R THEN 80
50   IF G < R THEN PRINT "TOO LOW"
60   IF G > R THEN PRINT "TOO HIGH"
70   GOTO 30
80   PRINT "CORRECT!"
```

The sign function SGN is useful in determining the sign of a number. SGN(X) returns -1, if X is negative; +1, if X is positive; and 0, if X is zero.

The absolute value function ABS can be used to drop the minus sign in a number. ABS(X) equals X, if X is positive; but if X is negative, ABS(X) gives -X a positive number. For instance, ABS(3) is 3 and ABS(-3) is -(-3), or +3. SQR, as discussed above, is the square root function.

The trigonometric functions SIN, COS, and TAN correspond to their usual definitions. This is illustrated in Figure 1. If a, b, and c are the lengths of the sides of a right triangle, and A is the angle in radians opposite the leg a, the SIN(A) is the ratio a/c. Similarly, COS(A) = b/c, and TAN(A) = a/b. A radian is a measure of angle chosen so that 2π radians is 360°. Equivalently, 1 radian is 57.3°, and one degree is 0.01745 radians.



$$\sin(A) = \frac{a}{c}$$

$$\cos(A) = \frac{b}{c}$$

$$\tan(A) = \frac{a}{b}$$

Figure 1. The Definitions of Trigonometric Functions.

EXAMPLE B: FUNCTIONS IN BASIC.

Given: BASIC in immediate mode.

Find: The value of x where:

$$x = 3*\sin(6y-z)$$

$$y = \left|\sqrt{\tan(z-r)} - \cos(3rz-1)\right|$$

$$z = 3.4$$

$$r = 2.71$$

Note: | x | is a common shorthand for "the absolute value of the expression x."

Solution: The equations can be typed almost as given, but in the reverse order so they can be evaluated:

```
R = 2.71
Z = 3.4
Y = ABS((SQR(TAN(Z-R))) - COS (3*R*Z-1))
PRINT 3*SIN(6*Y-Z)
```

The result is 2.98164. If the long expression is too confusing, it can be evaluated in parts by defining new, intermediate variables:

```
A = TAN(Z-R)
B = SQR(A)
C = COS(3*R*Z-1)
D = B-C
Y = ABS(D)
```

EXAMPLE C: GRAPHING A FUNCTION.

Given: The function Y = sin(X)cos(20X).

Find: A program that will graph this for X from -6 to 6 on the screen.

Solution: Remember that the screen coordinates run from -64

Example C. Continued.

to 64 and that the smallest step size is 0.2. If X runs from -6 to 6, this can be multiplied by 10 to get the horizontal screen coordinate. Then the graph will almost fill the screen as X goes from -6 to 6.

The function is evaluated in line 50. Its largest value will be 1; so Y is multiplied by 45 in line 60 to get the vertical screen coordinate.

```
 5 SCROLL OFF
10 ERASE
20 PEN UP
30 FOR I = -600 to 600
40 X = 0.01*I
50 Y = SIN(X)*COS(20*X)
60 JUMP TO 10*X, 45*Y
70 PEN DOWN
80 NEXT X
```

SUBROUTINES

A subroutine is part of a program that accomplishes a specific task that may be needed at more than one point in a program. Figure 2 illustrates this concept. When the main program reaches point A, a particular job must be done, such as calculating a special function or drawing a figure. This job is accomplished in a subroutine starting at C. When the subroutine reaches D, it is finished and should return to A so the main program can continue.

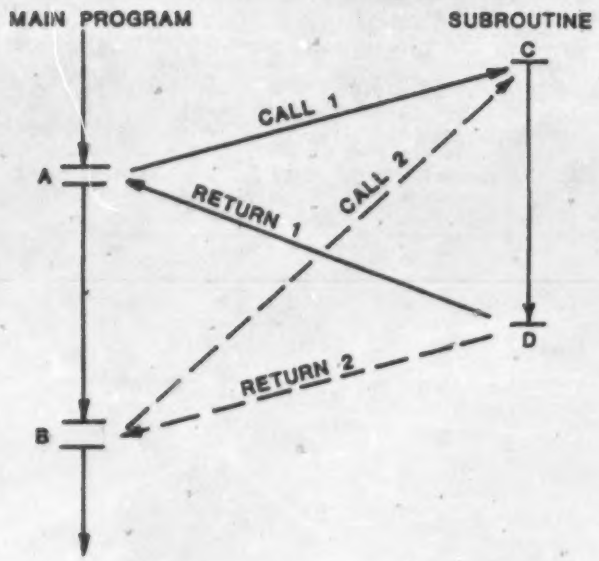


Figure 2. Execution Flow of a Program Calling a Subroutine Twice.

When the main program reaches B, the subroutine is needed again; so control is again passed to C. This time, when D is reached, control should return to B so the main program can continue where it left off this time.

Subroutines require two new BASIC instructions; GOSUB to go to the subroutine and RETURN to mark its end.

The act of passing control to a subroutine is named a call. In BASIC, subroutines are "called" with the following instruction:

```
GOSUB N
```

N is the line number at the beginning of the subroutine. N can be an expression if its value is a line number.

The subroutine starts at N and ends with the special instruction RETURN. RETURN instructs BASIC to go back to the calling program and resume executing the instruction following the GOSUB instruction.

Subroutines should not be at the beginning of a program. As a result, they are usually given large line numbers so they will follow the main program, as in the example below (line 1000 was chosen).

With the subroutine following the main program, there must be a way to tell BASIC where to stop. With no end indication, BASIC will continue beyond the end of the main program and start executing the subroutine as though it were more of the main program. When BASIC encounters a RETURN statement, it will know something is wrong because there is nowhere to return. It will stop and print out "CONTROL STACK ERROR".

To alleviate these problems, there is an instruction STOP which marks the end of the main program and tells BASIC to stop.

The following is a subroutine that can draw a square of size L:

```
1000 PEN DOWN
1010 FOR I = 1 TO 4
1020 MOVE L
1030 TURN 90
1040 NEXT I
1050 PEN UP
1060 RETURN
```

This subroutine might be used in the following main program to draw two concentric squares:

```
10 PEN UP
20 HOME
30 ERASE
40 L = 20
50 GOSUMB 1000
60 JUMP TO -10,-10
70 L = 40
80 GOSUB 1000
90 STOP
```

Instruction 40 sets the square size to 20. The instruction with line number 40 calls the subroutine, which draws the square. Then the turtle is moved to the start of the next square and the square size is set at 40. That square is drawn with the subroutine call in line 80.

EXAMPLE D: ARCOS SUBROUTINE.

Given: The subroutine below that calculates the value of a fraction is known as the inverse cosine (arcos). This subroutine was needed in the solar collector program in Module MO-05.

```
1000 REM THIS SUBROUTINE CALCULATES ARCOS(X)
1010 REM THE VALUE IS RETURNED IN Y
1020 IF X > 1 THEN 1130
1030 IF X < -1 THEN 1130
1040 IF X = 1 THEN 1150
1050 IF X = -1 THEN 1170
1060 IF X > 0.5 THEN Y = SQR(2*(1-X)
1070 IF X < -0.5 THEN Y = 3.1416-SQR(2*(1+X)
1080 Y = 1.5708*(1-X)
```

Example D. Continued.

```
1090 D = COS(Y)-X
1100 IF D <0.001 THEN RETURN
1110 Y = Y-D/SIN(Y)
1120 GOTO 1090
1130 :"ARGUMENT OF ARCOS OUT OF RANGE"
1140 RETURN
1150 Y = 0
1160 RETURN
1170 Y = 3.1416
1180 RETURN
```

Find: Write a program that asks for a number and prints the inverse cosine of that number.

Solution: The main program asks for the argument in line 10, calculates the inverse in line 20 and prints the result in line 30. The subroutine must follow this program; so line 40 is needed to stop execution.

```
10 INPUT "GIVE ARGUMENT FOR THE INVERSE FUNCTION": X
20 GOSUB 1000
30 PRINT "THE INVERSE COSINE OF": X, "IS": Y
40 STOP
```

EXERCISES

1. Suppose X and Y are both indexed variables, each containing 100 values that correspond to the x, y screen coordinates of 100 points that is to be graphed. Write a program that draws a graph that consists of a line connecting successive points - in order - from the first to the last.
2. Determine the effect of each of the following programs:

   a.
   ```
   10 FOR I = 0 to 2 STEP 0.1
   20 JUMP TO 10*I, I*I*I
   30 NEXT I
   ```

   b.
   ```
   10 DIM X (10)
   20 FOR I = 1 TO 10
   30 X(I) = I -SQR(I)
   40 NEXT I
   ```

LABORATORY MATERIALS

A disk-based microcomputer CP/M and BASIC.
1 full-size floppy disk.

LABORATORY PROCEDURES

1. Enter and run GUESS.

   GUESS, the number-guessing program listed in Table 4 illustrated conditional statements and functions. Enter it into the computer, run it and save it on the disk.

2. Modify GUESS.

   GUESS is fairly simple as it is written. Modify it in the following ways:

   a. It prints "CLOSE" if the guess is wrong but within 3 of the correct value.
   b. The range of possible numbers is 1 to 100 instead of 1 to 1000.
   c. The program starts over with a new random number after it is played once. Record the complete programs in the Data Table.

3. Graph functions.

   Example C contains a convenient graphing program. Enter and execute the program as written. Name it GRAPH and save it. Describe the output on the screen.

   Now use the ideas in that program to graph other functions. The range of X values is set in line 30. The function is evaluated in line 40. These values cannot be graphed directly, however, because the screen coordinates are fixed. Line 50 multiplies the X and Y values so they fill up the screen. To see the importance of this, try running the program by just plotting the X and Y values directly. This can be done in the program by substituting the following line:

```
50 JUMP TO X,Y
```

Describe the results of running this program as modified. Describe the effect of running the program with the following inserted:

```
50 JUMP TO 5*X + 30, 20* Y + 20
```

The multipliers 10 and 45 used in the original line 50 were chosen to fit the needs of the function graphed. To graph other functions, other multipliers may be needed. The constants 30 and 20 in the last example move the graph on the TV. These can be used to move a graph if the origin of the graph is not wanted in the center of the screen.

Graph each of the following functions so that the screen is filled. Record the program and describe the results in Data Table.

$Y = \frac{SIN(X)}{X}$	for X between -6 and 6
Y = 1+COS(X)	for X between -6 and 6
Y = SQR(X)	for X between 0 and 10

4. Use the autoscaling graphing program.

The problems encountered above (modifying the program to fill the screen) can be solved automatically using the AGRAPH - an automatic scaling graphing program. Load AGRAPH from the master disk and store it on your disk. AGRAPH is in BASIC, so load BASIC and call for AGRAPH as an old program.

When run, the program is self explanatory. Run it for X between 0 and 10 and describe the function it graphs.

To change graphs, the function must be changed. The function is at line 300. Any statement that deter-

mines H, the height or Y coordinate, in terms of X can be used.

Change line 300 to graph functions:

$$H = COS(X) + COS(2X) + COS(3X)$$

for X between -3 and 3

$$H = \frac{1}{(X-4)^2 + 0.1}$$

for X between 2 and 6

Describe the results seen on the TV screen.

DATA TABLE

STEP 2: GUESS

Results of run: Number chosen _____ Number of guesses required.

Modifications:

1. To print CLOSE: _____

2. To change the range: _____

3. To continue playing: _____

STEP 3: GRAPHING

Describe the result of running GRAPH: _____

Describe the results when line 50 is 50 JUMP TO X,Y: _____

Explain why this is seen: _____

Describe the results with 50 JUMP TO 5*X+30, 20*Y+20: _____

Explain: _____

Sketch the graph of the following functions:

$\frac{SIN(X)}{X}$

$-6 < X < 6$

Data Table. Continued.

1+COS(X

-6 < X < 6

Y = SQR(X)

0 < X < 10

STEP 4: AGRAPH

Sketch of graph as supplied:

What function is this?

Sketch of:

H=COSX + COS 2X
+COS 3X

-3 < X < 3

Sketch of:

$$H = \frac{1}{(X-4)^2 + 0.1}$$

2 < X < 6

REFERENCES

Albrecht, Bob; Finkel, Leroy; and Brown, Jerald R. BASIC for Home Computers. New York: John Wiley & Sons, Inc., 1978.

BASIC User's Manual. Cambridge, MA: TERC.

Dwyer and Critchfield. BASIC and the Personal Computer. Reading, AM: Addison-Wesley Publishing Company, 1979.